Encodix

Encodix - Release 1.0.131
Dec 27th, 2012

Dafocus
http://www.dafocus.com/

1 INTRODUCTION 5

1.1 ENCODIX PHILOSOPHY 5
1.2 ENCODIX MODULES 5
1.3 BUILD PROCESS 5
1.4 HOW IT WORKS 6

2 INSTALLING AND USING ENCODIX 8

2.1 INSTALLATION 8
2.2 USAGE 8

3 MESSAGES DESCRIPTION FILE 9

3.1 BASIC CONCEPTS 9
3.1.1 MESSAGE 9
3.1.2 MESSAGE SETS 10
3.1.3 INFORMATION ELEMENT PARTS 10
3.2 GENERAL INFORMATION 10
3.2.1 INCLUDE 10
3.2.2 COMMENTS 10
3.3 BASIC TYPES 11
3.4 ROOT SUB-MODULE 11
3.4.1 ROOT BNF GRAMMAR 11
3.5 COMMON PARTS 11
3.5.1 BNF GRAMMAR 11
3.6 SIGNAL LIST DECLARATIONS 12
3.6.1 BNF GRAMMAR 12
3.6.2 EXAMPLE 12
3.7 MESSAGE SET DECLARATIONS 12
3.7.1 BNF GRAMMAR 12
3.7.2 EXAMPLES 12
3.8 TLV BASIC MODULE 12
3.8.1 BNF GRAMMAR 13
3.8.2 EXAMPLES 13
3.9 TLV BASE MESSAGES 13
3.9.1 MESSAGE BNF GRAMMAR 13
3.9.2 EXAMPLES 14
3.9.3 MESSAGES WITH L2 PSEUDO LENGTH 17
3.9.4 TS 23.040 MESSAGES 18

4 INFORMATION ELEMENTS 20

4.1 BIT-FIELDS DECLARATIONS 22
4.1.1 BIT-FIELDS BNF GRAMMAR 22
4.1.2 EXAMPLES 23
4.2 0/1-EXT BIT-FIELDS 25
4.2.1 0/1-EXT BIT-FIELDS GRAMMAR 25
4.2.2 EXAMPLES 26
4.3 SEQUENCE DECLARATIONS 27
4.3.1 SEQUENCE BNF GRAMMAR 27
4.4 CUSTOM INFORMATION ELEMENTS 28
4.4.1 CUSTOM IE BNF GRAMMAR 28
4.4.2 ENCODING AND DECODING FUNCTIONS 29
4.4.3 EXAMPLES 30
4.4.4 OPTIONAL PARAMETERS 31
4.5 SUB-FIELDS 32

4.5.1 BNF GRAMMAR 32
4.5.2 EXAMPLES 32
4.6 TLVSET SUBFIELDS 32
4.7 DECODING MULTIPLE PROTOCOL LAYERS 33
4.7.1 EXAMPLES 33
4.8 CSN.1 MODULES 38
4.8.1 BNF GRAMMAR 38
4.8.2 NOTES 38
4.8.3 DATA TYPE GENERATION RULES 38
4.8.4 RULES FOR STRUCTURED DATA 39
4.8.5 HANDLING INFINITE REPETITIONS 42
4.8.6 CUSTOM RULES 42
4.8.7 THE SLAVE KEYWORD 47
4.8.8 THE INCOMING/OUTGOING KEYWORDS 47
4.8.9 THE TYPE-ONLY KEYWORD 47
4.8.10 THE MANUAL-XXX KEYWORDS 47
4.8.11 OTHER EXAMPLES 48
4.9 DECODED FIELDS VALIDATION 48

5 C OUTPUT 50

5.1 GENERATED STRUCTURES 50
5.1.1 BOOLEANS 50
5.1.2 OPTIONAL FIELDS 50
5.1.3 BINARY FIELDS 50
5.1.4 ARRAYS 51
5.1.5 UNIONS 51
5.1.6 AUTOMATIC SORTING OF STRUCTURES 51
5.2 DYNAMIC DATA MODULE 52
5.2.1 ACTIVATING THE DYNAMIC GENERATION 52
5.2.2 FUNCTIONS USED FOR DYNAMIC ALLOCATION 52
5.2.3 GENERAL RULES FOR DYNAMIC TYPES 52
5.2.4 DYNAMIC STRUCTURES 53
5.2.5 DYNAMIC UNIONS 53
5.2.6 DYNAMIC SEQUENCES 54
5.2.7 DYNAMIC BINARY VARIABLES 55
5.3 GENERATED FUNCTIONS 56
5.3.1 ENCODING FUNCTIONS 56
5.3.2 DECODING FUNCTIONS 57
5.3.3 SET-DECODING AND ENCODING FUNCTIONS 57
5.3.4 MATCH FUNCTIONS 58
5.4 DECODING BADLY FORMED MESSAGES 58
5.4.1 GENERAL RULES 58
5.4.2 MESSAGE TOO SHORT 58
5.4.3 TLV FIELD WITH LENGTH OUT OF RANGE 58
5.4.4 UNKNOWN OR UNFORESEEN MESSAGE TYPE 58
5.4.5 UNKNOWN AND UNFORESEEN IES 59
5.4.6 MISSING MANDATORY IES 59
5.4.7 CONDITIONAL IE ERRORS 59
5.4.8 DEFINED RETURN VALUES 59

6 SDL OUTPUT 61

6.1 THE GENERATED SDL-PR FILE 61
6.1.1 SDL NEWTYPE CONVENTIONS 61
6.1.2 ENCODE/DECODE OPERATORS 61

7 VISUAL BASIC INTEGRATION 62

7.1 ACTIVATION OF VISUAL BASIC GENERATOR 62
7.2 USAGE OF THE VISUAL BASIC INTEGRATION 62

7.2.1 DECODING 62
7.2.2 ENCODING 62

8 MULTI-FILE 63

8.1 DECLARATION OF SOURCE FILES 63
8.2 EXTERNAL DATA TYPES 64
8.2.1 SPLITTING CSN.1 64
8.3 BATCH FILE 64

9 DUMP MODULE 65

9.1 PRINCIPLES 65
9.1.1 DUMP ABSTRACTION 65
9.2 USING DUMP MODULE 66
9.2.1 GENERATING, COMPILING AND LINKING 66
9.2.2 UPDATING THE SOURCE FILES TO INTRODUCE CUSTOM DUMPS 66

10 ACCESS MODULE 67

10.1 BASIC CONCEPTS 67
10.2 THE ACCESS CLASSES 68
10.2.1 TYPE DESCRIPTION METAMODEL 68
10.2.2 DATA DESCRIPTION METAMODEL 68
10.2.3 USAGE 69

11 EXTENDED CONFIGURATION 70

11.1 TCL SHORT SUMMARY 70
11.2 CONFIGURATION VARIABLES 70
11.2.1 GENERATION OF SDL SUPPORT FILES 70
11.2.2 GENERATION OF SDL ENCODE/DECODE OPERATORS 70
11.2.3 SDL NEWTYPES PREFIX 70
11.2.4 PASSING EXTRA PARAMETERS TO ENCODE/DECODE FUNCTIONS 71
11.2.5 DOING EXTRA ACTIONS BEFORE SENDING AN SDL SIGNAL 71
11.2.6 GENERATION OF "TIGHT" C STRUCTURES 71
11.2.7 DEFINING GENERATED FILE NAMES 71
11.2.8 OTHER VARIABLES 72
11.2.9 VISUAL BASIC CONFIGURATION VARIABLES 74
11.2.10 GENERATED C TYPES CONFIGURATION VARIABLES 74

12 LICENSE 75

12.1 LICENSE TYPES 75
12.2 LICENSE SERVER DAEMON 75
12.2.1 INSTALLING OBJLICD 75
12.2.2 OPTIONS 76
12.2.3 CHALLENGE/RESPONSE METHOD 76

13 CHANGES 78

1 Introduction

1.1 Encodix philosophy

Encodix is a tool designed to meet all the encoding and decoding needs in telecommunication software.
Encodix is not a C library: it is a complete code generator which creates optimised source code that can be integrated with any environment.
One of the most appreciated key points of Encodix is its ability of accept user-friendly inputs and outputs. Input files are written using a custom syntax which resembles ITU's official documents: when official descriptions are informal (free-text, for example), a specific formal language representing the same concepts is supported by an Encodix module.

1.2 Encodix modules

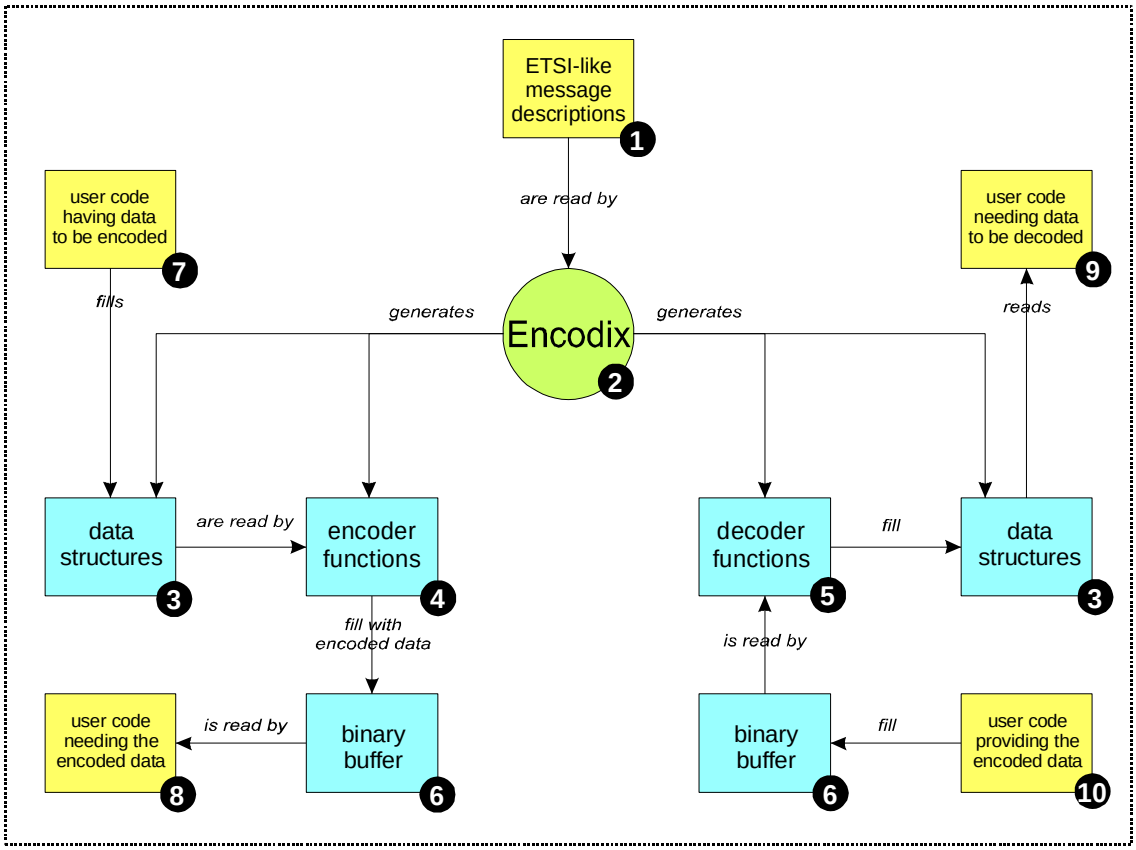
The TLV Encodix module is designed to support documents based on GSM 04.07 chapter 11. Information elements can be further described at bit level with this module. All formats are supported directly or by minimal C coding.
CSN.1 information elements are supported by a separate Encodix module.

1.3 Build process

Standard Encodix module reads an input file, which syntax is specified in following paragraphs, and generates several output files.
Some are C files, other are SDL-PR files: they must be included in the respective C/SDL projects.

1.4 How it works

The diagram below summarizes how Encodix works. Several advanced features and details are hidden for simplicity.



N.	Description

1	Encodix users write the messages description files. These files are expressed by using a specific formal syntax which resembles ETSI specification documents: this makes really easy to write and maintain message description files. In this way, all messages and information elements are specified down to bit-level detail.
	
	In this example, we declare a simple GSM-04.07 message:
	
	`gsm-0407 MY_MESSAGE {`
	`  ProtocolDiscriminator = 0101;`
	`  MessageType           = 1X001101;`
	
	`  4- foo    M  TV     1  integer;      /* 4 bit integer */`
	`  23 bar    O  TLV    3  MY_SUB_FIELD; /* see below */`
	`}`
	`/* Custom sub field */`
	`bit-field MY_SUB_FIELD {`
	`  size: 1 octet;`
	`  bits 1-5 partA integer;`
	`  bits 6-8 partB integer;`
	`}`

2	Encodix is an executable which reads the messages description file (1). It produces some .h and .c files (.pr files if SDL output is activated).

3	The data structure file is a .h file generated by Encodix (2) which contains one C typedef per each message and information element defined in the messages description file (1). This file contains user-friendly, abstract structured data types which are completely unrelated to their encoding format.
	
	For the example above, it generates:
	
	`typedef struct _c_MY_SUB_FIELD {`
	`  int partA;`

	`  int partB;`
	`} c_MY_SUB_FIELD;`
	
	`typedef struct _c_MY_MESSAGE {`
	`  int foo;`
	`  c_MY_SUB_FIELD bar;`
	`  ED_BOOLEAN bar_Present;`
	`} c_MY_MESSAGE;`

4	Encodix (2) generates an encoding C function per each message. Each function of these receives a pointer to a structure (3) and a pointer to a buffer (6). After the execution of one of these functions, the contents of the input structure will be encoded in the output binary buffer which will be ready to be used. For MY_MESSAGE, it generates: `long ENCODE_c_MY_MESSAGE (char* Buffer, const c_MY_MESSAGE* Source);`
5	Encodix (2) generates an decoding C function per each message. Each function of these receives a pointer to a structure (3) and a pointer to a buffer (6). After the execution of one of these functions, the contents of the input structure will be filled with the values decoded from the input binary buffer. For MY_MESSAGE, it generates: `long DECODE_c_MY_MESSAGE (const char* Buffer, c_MY_MESSAGE* Destin, long Length);`
6	A char buffer must be provided when encoding and decoding in order to store the encoded version. See the examples below and above.

7/8	Once code is generated, users can encode messages.
	
	In this example we need to encode an instance of “MY_MESSAGE” by setting “foo” to 2 and writing into “bar.partA” value 3 and “bar.partB” value 4:
	
	`char Buffer [50];`
	`c_MY_MESSAGE myMsg;`
	`long LenInBits;`
	`myMsg.foo = 2;`
	`myMsg.bar.partA = 3;`
	`myMsg.bar.partB = 4;`
	`myMsg.bar_Present = ED_TRUE; /* Optional field: mark it as present! */`
	
	`LenInBits = ENCODE_c_MY_MESSAGE (Buffer, &myMsg);`
	
	`/* Now “LenInBits” contains the length of the encoded message expressed in bits. Buffer contains an encoded message */`
	
	We had to specify only sensible data, without having care of writing protocol discriminator, skip indicator, information elements tags and lenghts and bit alignments: all these tasks are performed by the Encodix generated function.

9/10	Same thing happens when we need to decode.
	
	In this example we need to decode a buffer containing an instance of “MY_MESSAGE”:
	
	`/* Buffer is a const char* containing the message */`
	`/* Len is a long containing the buffer length from the layer below */`
	`c_MY_MESSAGE myMsg;`
	`DECODE_c_MY_MESSAGE (Buffer, &myMsg, Length);`
	`/* Now myMsg contains the decoded data: use it somehow… */`
	`printf (“foo is %d\n”, myMsg.foo);`

2 Installing and using Encodix

2.1 Installation

Encodix is distributed as a ZIP file, usually named `Encodix_x.y.z.zip`, being `x.y.z` the version number.
The distribution ZIP file must be expanded in an empty directory maintaining the stored directory sub-tree.
NOTE: it is recommended to install each version of Encodix in its own empty directory and not overwriting an existing version of Encodix.

2.2 Usage

In order to run Encodix, users should prepare a *messages description file*, which is what Encodix takes as a source file; also, users should create a directory where generated files will be placed (*output directory*).
The command line required to run Encodix is:

```
<ed>\bin\codegenix <ed>\common\Encodix.tcl <sf> <od> [config=<cfg>]
```

where:

<ed>	is the directory where Encodix has been installed;
<sf>	is the message description file;
<od>	is the output directory;
<cfg>	is an optional configuration file described in chapter "Extended configuration"

For example, if we have installed Encodix under `C:\Encodix` and we want to process file `test.src`, which is in the current directory, placing the output in `C:\out`, we should run:

```
C:\Encodix\bin\codegenix C:\Encodix\common\Encodix.tcl test.src
C:\out
```

3 Messages description file

The "Messages description file" is a plain ASCII file. Features are divided in sub-modules. Each sub-module has an independent grammar and is described in a chapter.

3.1 Basic concepts

A message description file contains declaration of the following elements:

- messages
- message sets
- information element parts
- signal lists

3.1.1 Message

A "Message" is an entity made of:

- a signature, which allows to recognize the message inside a binary stream;
- a data structure.

> *For example, a TS 24.008 "LOCATION UPDATING REQUEST" can be expressed with an Encodix "message". In fact a "LOCATION UPDATING REQUEST" has its own signature (its protocol discriminator and its message type) and a data structure (its information elements).*

Messages have to be declared as part of a *message family*. Each message family has a specific way of recognizing messages. Normally, these families are named after the first ETSI document describing that message family. The supported families are:

Name	**Aka**	**Signature**
gsm-0407	gsm-0408, tsm-0408	• 4 bit protocol discriminator • 4 bit skip indicator or transaction ID (no ext. trans. ID) • 8 bit message type
gsm-0407-plen	gsm-0408-plen, tsm-0407-plen	Like a "gsm-0407", but before protocol discriminator a "L2 Pseudo Length" field is expected (see GSM 04.08, 10.5.2.19)
gsm-0407-xtid	gsm-0408-xtid, tsm-0407-xtid	Like a "gsm-0407", but transaction ID can be 4 or 12 bits long (see TS 24.007, 11.2.3.1.3).
abis	sbs-abis	• 8 bit message discriminator • 8 bit message type
gsm-0808		• 8 bit message type
gsm-0816		• 8 bit PDU type
gsm-0460	tsm-0460	• 6 bit message type
TS-23.040		• 2 bit message type, specific for TS 23.040
tcp-lind		• 16 bit Length indicator • 4 bit protocol discriminator • 4 bit skip indicator or transaction ID (no ext. trans. ID) • 8 bit message type • 8-16 bit extensible length for TLV fields Used in UMA Protocols Stage 3
udp		• 8 bit message type • 8-16 bit extensible length for TLV fields Used in UMA Protocols Stage 3
Id2		• 2 bit message type

3.1.1.1 Generated output for messages

Given a message, Encodix generates (SDL parts are generated only if the SDL module is installed and active):

- a C structure representing the message data structure; this structure does not contain any encoding information like protocol discriminator, message type, IE tags or lengths;
- a SDL structure representing the message data structure;

- an SDL signal transporting the above structure;
- a C encoding function, which translates from the C structure to a binary buffer;
- a C decoding function, which translates from the binary buffer to the C structure;
- a recognizing function, which reads a binary buffer and returns true if the contained message is the associated one.

3.1.2 Message sets

Messages can be grouped inside a "message set". Message sets have the following characteristics:

- all messages of a given message set must be homogeneous (i.e. part of the same message family);
- messages must avoid ambiguity (i.e. two messages of the same message set are not allowed to have the same signature);
- a message can be part of multiple message sets.

The main purpose of message sets is to create *recognisers*. A *recogniser* is a function which receives a binary buffer, analyses it and tells which message is inside it.

3.1.2.1 Generated output for message sets

Given a message, Encodix generates (SDL parts are generated only if the SDL module is installed and active):

- A C tagged[1] union containing data structures of all the messages participating to a message set;
- A message set decode function, with takes a binary buffer, recognizes it and decodes it in the tagged union.
- A *Sdl2Buffer* function, which takes an SDL signal and converts to a binary buffer;
- A *DecodeAndSend* function, which takes a binary buffers, decodes it and sends it as signal inside the SDL system.

3.1.3 Information element parts

Information elements can be described with simple types, like binary buffer or integer. But, often, information elements should be divided into sub-parts.
There are several ways of declaring information element: see chapter 3.9.4.

3.1.3.1 Generated output for information elements

Given an information element, Encodix generates (SDL parts are generated only if the SDL module is installed and active):

- a C structure representing the information element data structure;
- a SDL structure representing the information element data structure;
- a C encoding function, which translates from the C structure to a binary buffer;
- a C decoding function, which translates from the binary buffer to the C structure;

3.2 General information

The following information apply to any Encodix module.

3.2.1 Include

In every Encodix source file, other sub files can be included by inserting:

```
`INCLUDE xxx.src`
```

where single quotes are ASCII 96 and *xxx.src* is the file we wish to include.

3.2.2 Comments

Comments can be inserted in the source file. Supported comments are like the examples below:

[1] A "tagged union" is a C structure containing a `union` and an `enum` which tells which union field is valid.

```
/* this is a comment */
# this is another comment until end of line
```

3.3 Basic types

There are some basic data types that can be applied wherever a type is expected.

Type	Description
integer	Becomes an int in C and an integer in SDL
boolean	Becomes an int in C and an boolean in SDL
octet	Becomes a char in C and an octet in SDL
binary	Becomes a char[] in C and an octet_string in SDL
octet_array	Becomes a char[] in C and a CArray(octet) in SDL
void [default=n]	This field is not interpreted: therefore, no fields are generated in data structures, except if the field is optional or conditional: in such a case, a dummy field is created just to test its presence. If default=n is specified, where n is an integer value, then when encoding that default value will be written to the associated field.

3.4 Root sub-module

This is the root part of the input file.

3.4.1 Root BNF grammar

Root ::= {SignalListDeclLine | MessageSetDeclaration | Message | BitField | Sequence | ST_TypeDef | CustomLengthDeclaration }

3.5 Common parts

These are common declarations used by some other modules

3.5.1 BNF grammar

ABIS ::= ("sbs-abis" | "abis")

GSM0407 ::= ("gsm-0408" | "gsm-0407" | "tsm-0408")

GSM0407PLEN ::= ("gsm-0408-plen" | "gsm-0407-plen" | "tsm-0408-plen")

GSM0407XTID ::= ("gsm-0408-xtid" | "gsm-0407-xtid" | "tsm-0408-xtid")

GSM0808 ::= "gsm-0808" | "sm-rl"

GSM0460 ::= "gsm-0460"

FT_FieldType ::= ["force"] ("void" "default" "=" Numeric:Default | “void” | “integer” | “binary” | “boolean” | “octet_array” | “octet” | [“body” “of”] <Identifier:otherType> FT_ValidationCode ((["enco" <CCode:encoPars>] ["deco" <CCode:decoPars>]) | ["encodeco" <CCode:encodecoPars>])
)

FT_Size ::= Natural:sizeInBits ("bits" | "bit" | "octets" | "octet" | "bytes" | "byte")

FT_ValidationCode ::= ["validator" <CCode:validationCode> | "valid" "if" <CCode:validationExpression>]

CustomLengthDeclaration ::= "custom-len-decl" Identifier:custLenName "encode" CCode:custLenEncode "decode" CCode:custLenDecode {";"}

3.6 Signal list declarations

Signal lists are a SDL concept that allows grouping of signals (messages) under a single name. This permits simple declaration of signals on signal routes and channels.
Encodix provides automatic signal list creation by using the following statements.

3.6.1 BNF grammar

SignalListDeclLine ::= "`declare`" "`signallist`" SignalListDecl [{"`,`" SignalListDecl}] "`;`"

SignalListDecl ::= <Identifier:sigListName> ("`IN`" | "`OUT`")

3.6.2 Example

```
declare signallist TestIn IN, TestOut OUT;
```

3.7 Message set declarations

Message sets are sets of homogeneous messages. All messages participating to a message set must have an unique identity. For each message set, a specific recogniser function will be created.

3.7.1 BNF grammar

MessageSetDeclaration::= "`declare`" (GSM0407 | GSM0808 | GSM0460 | ABIS |
GSM0407PLEN | GSM0407XTID)
["`incoming`" | "`outgoing`"]
"`message`" "`set`" <Identifier:msgSetName>
["`(`" MsgSetParam {"`,`" MsgSetParam} "`)`"] ["`;`"]

MsgSetParam ::= <Identifier:paramType> <Identifier:paramName>

3.7.2 Examples

Example 1: normal or parametric message sets.
In this example three message sets are declared. Message set `MyMsgSet3` will be parametric, i.e. signals which are part of this message set will have three parameters (data parameter as usual, `p1` and `p2`) instead of one.

```
declare gsm-0407 message set MyMsgSet1;
declare sbs-abis message set MyMsgSet2;
declare gsm-0407 message set MyMsgSet3 (TType1 p1, TType2 p2);
```

Example 2: excluding generation of encode or decode functions
In order to optimise generated code or to avoid conflicting recognisers, it is possible to declare “outgoing” or “incoming” message sets:

```
declare gsm-0407 incoming message set InOnly;
declare gsm-0407 outgoing message set OutOnly;
```

In this case, no encoding functions will be generated for messages registered in “InOnly”; `Sdl2Buffer` functions are also removed.
Messages registered in “OutOnly” message set don’t get any decoding function generated; all recognisers are also removed.

3.8 TLV Basic module

This module gives basic support for 04.07 information elements.

3.8.1 BNF grammar

TLV_InformationElement ::= ((TLV_Type1_V1 TLV_Type1_V2)| TLV_Type1_V | TLV_Type2 | TLV_Type3_V | TLV_Type3_TV | TLV_Type4_LV | TLV_Type4_TLV | "**EndOfL2Length**" ";")

TLV_InformationFooter ::= FT_FieldType [<CCode:condCode>] [TLV_CustomLen]";"

TLV_InformationHeader ::= <Identifier:IEName> TLV_Presence

TLV_Presence ::= ("T" | "0" | "C")

TLV_SizeRange ::= <Value:minOctets> "-" <Value:maxOctets> | ">=" <Value:minOctets> | <Value:minOctets>

TLV_Tag4 ::= <IEI4:Tag>

TLV_Tag8 ::= <IEI8:Tag>

TLV_Type1_V ::= TLV_Tag4 TLV_InformationHeader "TV" "1" TLV_InformationFooter

TLV_Type1_V1 ::= TLV_InformationHeader "V" "1/2" TLV_InformationFooter

TLV_Type1_V2 ::= TLV_InformationHeader "V" "1/2" TLV_InformationFooter

TLV_Type2 ::= TLV_Tag8 TLV_InformationHeader "T" "1" TLV_InformationFooter

TLV_Type3_TV ::= TLV_Tag8 TLV_InformationHeader "TV" <Value:maxOctets> TLV_InformationFooter

TLV_Type3_V ::= TLV_InformationHeader "V" <Value:maxOctets> TLV_InformationFooter

TLV_Type4_LV ::= TLV_InformationHeader ("LV" | "L2V") TLV_SizeRange TLV_InformationFooter

TLV_Type4_TLV ::= TLV_Tag8 TLV_InformationHeader ("TLV" | "TL2V") TLV_SizeRange TLV_InformationFooter

TLV_Type5_TV ::= TLV_Tag8 TLV_InformationHeader "TV" TLV_SizeRange TLV_InformationFooter

TLV_Type5_V ::= TLV_InformationHeader "V" TLV_SizeRange TLV_InformationFooter

TLV_CustomLen ::= “custom-len” (identifier:customLenName) | ("encode " CCode:lenEncode "decode" CCode:lenDecode)

3.8.2 Examples

For complete examples, see example section of “TLV base messages”.

3.9 TLV base messages

This sub-module allows description of primary level message. It allows to describe the typical TLV table with information elements, protocol discriminator and so on.

3.9.1 Message BNF grammar

Message ::= (GSM0407 | GSM0808 | GSM0460 | ABIS | GSM0407PLEN | GSM0407XTID) <Identifier:msgName> "{" [InList ";"][InSet] MessageDescriptor ["HandleTransactionIdentifier" [";"]] ["L2PseudoLength" "=" <Natural:L2PseudoLength> [";"]] ["PLenAutoFill" "=" <Natural:pLenAutoFill> [";"]] {TLV_InformationElement | StandardField} [TLV_Type5_V | TLV_Type5_TV] "}" [";"]

MessageDescriptor ::= (
"ProtocolDiscriminator" "=" <Bin4:pd> */* Not with ABIS */*
| "MessageDiscriminator" "=" <Bin8:md> */* With ABIS only */*
) ";" "MessageType" "=" (<Bin8X:mt> | <Bin6:mt>)";"

InList ::= "in" ("signallist" | "signallists") InListName {"," InListName}

InListName ::= <Identifier:sigListName>

InSet ::= "in" "message" "set" InSetElement {"," InSetElement} ";"

InSetElement ::= <Identifier:name> ["as" Identifier:sigName [InList]]

3.9.2 Examples

Example 1 – basic 04.07 TLV messages
The following example declares a message in GSM-04.07 standard format. All types of information elements, from type 1 to type 5, are reported. Please refer to GSM-04.07 chapter 11 for further information.

```
#------------------
# EXAMPLE
#------------------
gsm-0407 TESTMSG_IN {
     in signallist TestIn;
     ProtocolDiscriminator = 0101;
     MessageType           = 1X001101;

      t1Va           M   V     1/2 octet;
      t1Vb           M   V     1/2 octet;
   4- t1TV           M  TV     1   integer;
   23 t2T            O   T     1  ;
      t3V            M   V     4  ;
   FF t3TV           M  TV     4  ;
      t4LV           M  LV     3-10;
   AB t4TLV          M  TLV    >=3;
   18 opt1           O  TV     5 integer;
      cond1          M   V     1 integer;
      condField      C   V     2 integer %{DATA.cond1 > 0}%;
      t5V            M   V     2-10;
}
```

Please note the following features:

- Protocol discriminator and message type are declared as found on ETSI documents; message type supports `bXbbbbbb` format too.
- Some fields have `integer` type: they will be expanded as integers; other fields will be extracted in binary form. Other types are shown in following examples.
- **Signal lists** – Declaration `in signallist` allows to specify in which SDL signal list this message should be included. Signal lists must have been declared with `declare singnallist` statement.
- **Conditional fields** – Untagged conditional fields can be recognized by a C expression. As stated by ETSI documents, conditions can be always fulfilled by reading values of the preceding fields. In the example, `condField` is conditional; its condition is that the field is present if `cond1` is greater than 0. This expression is reported writing C code between `%{…}%`; fields can be accessed by using `DATA`, as shown in the example.
  Inside those expressions is always defined either `ED_IS_DECODING` or `ED_IS_ENCODING`: these macros can be used with preprocessor directives to obtain two different expressions when decoding and encoding.
  From version 1.0.75, Encodix generates the conditional fields as “optional” if they are tagged and the condition is not explicitly stated. This feature can be disabled by setting `ED_CONVERT_UNSPECIFIED_CONDITIONS_TO_OPTIONAL` to 0.

- **Variable sized fields >=n** – Some fields have unlimited size (>=3, for example). In this case, a default size will be allocated. To avoid unnecessary usage of memory, a reasonable maximum value should be specified.

Example 2 – 04.07/24.007 messages with transaction identifier
This example shows a message having a transaction ID, has defined in GSM 04.07 chapter 11.2.3 or TS 24.008, chapter 11.2.3.1.3. Default behaviour is to have a *skip indicator* field set to 0; with the `HandleTransactionIdentifier` keyword, skip indicator is substituted by the transaction identifier. A declaration like the following is needed:

```
#------------------
# EXAMPLE MESSAGE
#------------------
gsm-0407 TRANSID_IN {
     in signallist TestIn;
     ProtocolDiscriminator = 0111;
     MessageType           = 10000011;
     HandleTransactionIdentifier;

  aa dummy                 M TV 2 integer;
  ...etcetera
```

In this case, declaration `HandleTransactionIdentifier` will create two extra fields in subsequent data structures:
- `TI_Value` – is an integer; contains bits 5-7;
- `TI_Flag` – is a Boolean; contains bit 8.

WARNING! If the message supports extendible transaction identifier (see TS 24.007, 11.2.3.1.3), i.e. when transaction identifier can be either 1/2 or 3/2 octets long, message must be declared "`gsm-0407-xtid`" instead of "`gsm-0407`".

Example 3 – 08.08 messages
Messages declared on GSM 08.08 document are basically standard TLV messages. The main difference is found on recognition, which is made thanks to first byte instead of first two bytes. Gsm 08.08 messages are declared as follows:

```
gsm-0808 GSM0808_IN {
     in signallist TestIn;
     MessageType           = 00000001;

  aa d1                M TV 5 integer;
}
```

Example 4 – Abis messages
Abis messages are standard TLV messages, but instead of skip indicator and protocol discriminator they have an 8-bits Message Discriminator.

```
sbs-abis ABIS_IN {
     in signallist TestIn;
     MessageDiscriminator  = 00000011;
     MessageType           = 00000001;

  aa d1                M TV   5 integer;
  12 d2                M TL2V 3-6 integer;
}
```

This example shows also usage of 16-bit `L` information elements, i.e. those non-standard TLV IEs that use 16 bit for length instead of 8. They can be expressed by using `L2` instead of `L`.

Example 5 – 04.60 messages
Messages declared on GSM 04.60 are mostly described in CSN.1; they have a 6 bit message type.

```
gsm-0460 GSM0460_OUT {
     in signallist TestOut;
     MessageType           = 110010;

     body                M V >=0 TBody0460;
}

csn.1 {
  <TBody0460> ::= < PAGE_MODE :bit(2)>
        < my test : < octet > >;
}
```

Example 6 – Placing a message in multiple message sets
It is allowed to place the same message in many message sets.
Encodix generates a single data structure for each message, but it generates a distinct SDL signal per each message set. For this reason, when specifying further participation to further message sets, it is necessary to declare an alternative SDL name for the new signal. It is necessary also to declare in which signal list this new signal must be included.

```
gsm-0408 AbisTransp1_IN {
     in signallist TestIn;
     in message set
        AbisTransportedSet,
        Gulp as Gulp_AbisTransp1_IN in signallist TestIn;
```

…etc…

Example 7 – The 'force' keyword
The "`force`" keyword just disables the size consistency check between the information element declared size and the contained data. In the example below, the fact that `Tcontents` is really 3 to 5 octets long is disabled:

```
gsm-0460 MyMessage {
     MessageType           = 110010;

     myInformationElement  M LV 3-5 force TContents;
}
```

3.9.3 Messages with L2 Pseudo Length

As stated in GSM 04.08 chapter 10.5.2.19, "*The L2 Pseudo Length information element indicates the number of octets following it in the message which are to be interpreted in the scope of the phase 1 protocol, i.e. the total number of octets (excluding the Rest Octets) for which T, V, TV, LV, or TLV formatting is used (reference Table 11.1/GSM 04.07).*"

NOTE! Encodix versions prior to 1.0.57 calculated automatically the L2 Pseudo Length value applying the rules of the above statement; however, several 3GPP messages have been found declaring a specific L2 Pseudo Length which does not conform to the rules above. Therefore, the syntax has been extended and the L2 Pseudo Length can now to be declared manually through the "`L2PseudoLength`" keyword.

Example 1 – Messages with automatically calculated L2 pseudo-length
In this example we let Encodix calculate the message length automatically according to the rule above. It will write in the L2PseudoLength field the size of the d1 information element plus the size of the Protocol Discriminator and the Message Type.

```
gsm-0407-plen PLEN_IN {
     in signallist TestIn;
     ProtocolDiscriminator = 0110;
     MessageType           = 00000001;

  aa d1                 M TLV 2-10;
  **EndOfL2Length**;
     IARestOctets       M  V  0-11;
}
```

Keyword "`**EndOfL2Length**`" marks the beginning of the rest octets that have to be excluded by L2 pseudo length accounting.

Example 2 – Messages with fixed L2 pseudo-length

In this example we force the L2PseudoLength to 12, no matter which is the length of the message:

```
gsm-0407-plen PLEN_IN {
     in signallist TestIn;
     ProtocolDiscriminator = 0110;
     MessageType           = 00000001;
     L2PseudoLength        = 12;

  aa d1                 M TLV 2-10;
     IARestOctets       M  V  0-11;
}
```

Keyword "`L2PseudoLength`" is used to force the L2 Pseudo Length value.

Example 3 – Messages with L2 pseudo-length and auto-fill
If IARestOctets are of no interest, auto-fill feature can be used. In this case, remainder part of a message is filled up to a given size with a sequence of the default value 0x2B.
In the following example, the message is filled of 0x2B characters until size reaches 15 octets, comprising the L2 Pseudo Length field itself, protocol discriminator and message type.

```
gsm-0407-plen AF_PLEN_IN {
     in signallist TestIn;
     ProtocolDiscriminator = 0010;
     MessageType           = 00000001;
     L2PseudoLength        = 12;
     PLenAutoFill          = 15;

  aa d1                 M TLV 2-4;
}
```

For example, if d1 contains "0x01, 0x02", 8 fill octets will be generated.

3.9.4 TS 23.040 messages

This format is specified in the 3GPP TS 23.040 standard. The expected grammar is:

```
GPP23040Message ::= "TS-23.040" <Identifier:msgName> "{" [InList ";"] [InSet] ("TP-
                    MTI"|"TP_MTI") "=" <Bin2:TP_MTI> ";" {GPP23040Entry} "}" [";"]

GPP23040Entry   ::= <Identifier:entryName> ("M"|"0" <CCode:OptionalExpression>)
                    (("o" |("i"|"I") | <Value:MinSize> ["-" Value:MaxSize>] ("o"|"i"|"I"))
                    |[<Value:MinSize>] "b" "[" <Value:BitPosition> "]")   FT_FieldType
                    ";"
```

Example:

```
TS-23.040 SMS_DELIVERY_REPORT_RP_ACK {
  in message set MS2SC_ACK;
  TP_MTI = 00;
  TP_UDHI  M        b [6] boolean; # See 9.2.3.23
  TP_PI    M        o TP_PI     ; # See 9.2.3.27
  TP_PID   O        o octet     ; # See 9.2.3.9
  TP_DCS   O        o TP_DCS    ; # See 9.2.3.10
  TP_UD    O   0-161o TP_UD     ; # See 9.2.3.24;
};
```

3.9.5 IEEE 802.16 messages

This format has been designed to support messages in the form expressed by specifications like IEEE 802.16 and its derivativexs.
The grammar is:

```
IEEE80216Message ::= "msg-802.16" empty <Identifier:msgName> "{" [InList ";"][InSet]
                     "MessageType" "=" <Numeric:messageType> empty ";"
                     {IEEE80216_BodyElement} "}" {";"}

IEEE80216IE ::= "ie-802.16" <Identifier:ieName> ["(" IEEE80216IE_Parameter {"," 
                     IEEE80216IE_Parameter} ")"] "{" {IEEE80216_BodyElement} "}"
                     {";"}

IEEE80216IEID ::= "ie-802.16-id" <Identifier:ieName> "{" "IEType" "="
                     <Numeric:ieType> ";" "IETypeSize" "="
                     <Numeric:ieTypeSizeInBits> ("bits" | "bit") ";"
                     {IEEE80216_BodyElement} "}" {";"}

IEEE80216IESET ::= "ie-802.16-ieset" <Identifier:ieSetName> "{"
                     {IEEE80216IESETIeName} "}" {";"}

IEEE80216TLV ::= "ie-802.16-tlv" <Identifier:tlvName> "{"
                     "IEI" "=" <Numeric:IEI> ";" "Length" "=" (<Numeric:Length> |
                     "variable") ";"
                     ["MaxLength" "=" <Numeric:MaxLength> ";"]
                     "Type" "=" FT_FieldType empty ";"
                     ["Custom" "=" <CCode:customSelector> ";"]
                     "}" {";"}

IEEE80216TLVGroup ::= "ie-802.16-tlv-group" <Identifier:tlvGroupName> "{"
                     ["max" "=" <Numeric:GroupMax> ";"]
                     {IEEE80216TLVName}
                     "}" {";"}

IEEE80216IE_Parameter ::= <Identifier:parameterName>

IEEE80216IESETIeName ::= <Identifier:ieName>

IEEE80216TLVName ::= <Identifier:tlvName>
```

```
IEEE80216_BodyElement ::= IEEE80216_Body_Padding | IEEE80216_Body_Fix |
                          IEEE80216_Body_IECall | IEEE80216_Body_If |
                          IEEE80216_Body_Loop | IEEE80216_Body_DoWhile |
                          IEEE80216_Body_Custom

IEEE80216_Body_Fix ::= <Identifier:name> (<Numeric:sizeInBits> ("bit" | "bits") |
                          "variable" | <CCode:customExpression>)
                          ["maxsize" <Numeric:maxSizeInBits> ("bit" | "bits")]
                          FT_FieldType ";"

IEEE80216_Body_If ::= "if" IEEE80216_Body_IfCondition empty "{"
                          {IEEE80216_BodyElement} "}"
                          {IEEE80216_Body_ElseIf}[IEEE80216_Body_Else]

IEEE80216_Body_IfCondition ::= (
                          "(" (<Identifier:fieldName>
                          ["[" <Numeric:bitToBeCompared> "]"]
                          ["==" <Numeric:value> {"," <Numeric:value>}])
                          ")"
                          | <CCode:customExpression>)

IEEE80216_Body_Else ::= "else" "{" {IEEE80216_BodyElement} "}"

IEEE80216_Body_ElseIf ::= "else" "if" IEEE80216_Body_IfCondition empty "{"
                          {IEEE80216_BodyElement} "}"

IEEE80216_Body_IECall ::= <Identifier:ieName> "(" [IEEE80216_Body_IECall_Param {","
                          IEEE80216_Body_IECall_Param}] ")" ";"

IEEE80216_Body_IECall_Param ::= <Identifier:fieldName> | <CCode:customCode>

IEEE80216_Body_Padding ::= (("padding" <Numeric:paddingBits> ("bits" | "bit")) |
                          "padding-to-octet") ";"

IEEE80216_Body_DoWhile ::= "do" <Identifier:name> ["max" <Numeric:maxValues>] "{"
                          {IEEE80216_BodyElement} "}" "while"
                          <CCode:customExpression> {";"}

IEEE80216_Body_Loop ::= "loop" <Identifier:name> (<CCode:customExpression> | "("
                          <Identifier:topValue> ")")
                          ["max" <Numeric:maxValues>] "{" {IEEE80216_BodyElement} "}"
                          {";"}

IEEE80216_Body_Custom ::= "custom" {"encode" <CCode:CEncode> | "decode"
                          <CCode:CDecode> | "encodeco" <CCode:CEncoDeco> |
                          IEEE80216_Body_Custom_Var} ";"

IEEE80216_Body_Custom_Var ::= ("encovar" empty | "decovar" empty | "encodecovar"
                          empty)(<Identifier:VarType> | <CCode:VarType>)
                          <Identifier:VarName>
```

3.9.6 16-bit TAGs

If tags are expressed as four exadecimal digits, they are implemented as 16-bit tags instead of 8-bit.

For example:

```
#------------------
# EXAMPLE MESSAGE
#------------------
gsm-0407 TEST_MESSAGE {
     in signallist TestIn;
     ProtocolDiscriminator = 0111;
     MessageType           = 10000011;

  0012 Field1                M TV 2 integer;
  1EFA Field2                M TV 2 integer;
  …etcetera
```

3.9.7 Custom ‘L’ field

Normally the ‘L’ field in TLV entries is encoded as a single octet containing the length, in octets, of the following ‘V’ data.
From version 1.0.125 Encodix allows the custom declaration of the ‘L’ field encoding.

This declaration can be done inline:

```
gsm-0407 EXAMPLE_MESSAGE {
     ProtocolDiscriminator = 0111;
     MessageType           = 10000011;

  aa d1 M TLV   >=2 binary custom-len
   encode %{MyLEncode (Buffer, &CurrOfs, TLV_Base, VLEN);}%
   decode %{MyLDecode (Buffer, &CurrOfs, &Len);}%;
…
```

Also, it can be declared once and just referenced:

```
custom-len-decl ABCDEF
   encode %{MyLEncode (Buffer, &CurrOfs, TLV_Base, VLEN);}%
   decode %{MyLDecode (Buffer, &CurrOfs, &Len);}%

gsm-0407 EXAMPLE_MESSAGE {
     ProtocolDiscriminator = 0111;
     MessageType           = 10000011;

  aa d1 M TLV   >=2 binary custom-len ABCDEF;
…
```

The encoding function
The encoding function must logically perform the following steps:

- It receives in `VLEN` the length in bits of the ‘V’ part that has been already encoded;
- the ‘V’ part has been encoded in `Buffer` starting at offset `TLV_Base`;
- the custom-length encoding function must move the data in `Buffer` to create the space needed to fit the length field;
- then it can encode the length field in `Buffer` at `TLV_Base`;
- finally, it must increase `CurrOfs` of the number of bits used to encode the length field.

The decoding function
The decoding function must logically perform the following steps:

- it receives the offset in `Buffer` where the length field starts in `CurrOfs`;
- it must decode the length data and write in (in bits) in `Len`;
- finally, it must increase `CurrOfs` of the number of bits used by the length field.

Example
In this example we implement a variable-length encoded L field.
The ‘L’ field is so formed:

- if length is <= 7F (i.e. it fits in 7 bits) it is encoded as a normal, one octet L-field (having the first it set to 0).

- if length is >= 80 (i.e. if it nees 8 or more bits), the encoding is: `1 <A:bit(l)> <len:octet(val(A))>`; in other words, the first octet has the most significant bit set to 1; the remaning 7 bits tell the number of following octets that contain the 'length' field.

```
/* ENCODE */
void MyLEncode (void* Target, int* CurrOfsInBits, int OffsetInBitsOfLengthField, int
LengthInBitsOfValue)
{
 int LengthOfTheLFieldExtensionInOctets;
 int LengthInOctetsOfValue;

 /* Convert the 'LengthInBitsOfValue' in 'LengthInOctetsOfValue'. Make sure */
 /* it is round to the nearest highest integer */
 LengthInOctetsOfValue = (LengthInBitsOfValue+7) >> 3;

 /* First, we see the length 'LengthInBitsOfValue' and we decide how many */
 /* octets we need */
 if (LengthInOctetsOfValue >= 0x1000000) {
  LengthOfTheLFieldExtensionInOctets = 4;
 }
 else if (LengthInOctetsOfValue >= 0x10000) {
  LengthOfTheLFieldExtensionInOctets = 3;
 }
 else if (LengthInOctetsOfValue >= 0x100) {
  LengthOfTheLFieldExtensionInOctets = 2;
 }
 else if (LengthInOctetsOfValue >= 0x80) {
  LengthOfTheLFieldExtensionInOctets = 1;
 }
 else {
  LengthOfTheLFieldExtensionInOctets = 0;
 }

 /* Move the data part to create the space needed to host the */
 /* variable sized length field */
 ED_MEM_MOVE (
  ((char*)Target)+(OffsetInBitsOfLengthField >> 3) +
     (LengthOfTheLFieldExtensionInOctets+1),
     ((char*)Target)+(OffsetInBitsOfLengthField >> 3), LengthInBitsOfValue >> 3);

 /* Encode the octets */
 if (LengthOfTheLFieldExtensionInOctets == 0) {
  EDIntToBits (Target, OffsetInBitsOfLengthField, LengthInOctetsOfValue, 8);
 }
 else {
  EDIntToBits (Target, OffsetInBitsOfLengthField,
                0x80 | LengthOfTheLFieldExtensionInOctets, 8);
  EDIntToBits (Target, OffsetInBitsOfLengthField+8,
     LengthInOctetsOfValue,
     LengthOfTheLFieldExtensionInOctets << 3);
 }
 (*CurrOfsInBits) += ((LengthOfTheLFieldExtensionInOctets+1) << 3);
}

/* DECODE */
void MyLDecode (const void* Source, int* OffsetInBits, int* LengthInBits)
{
 /* Verify the first bit. If set to 1 we have the length-length */
 /* format */
 if (EDBitsToInt (Source, *OffsetInBits, 1)) {
  int LengthOfLengthInBits = EDBitsToInt (Source, (*OffsetInBits)+1, 7) * 8;
  (*LengthInBits) = EDBitsToInt (Source, (*OffsetInBits)+8, LengthOfLengthInBits);
  (*OffsetInBits) += 8 + LengthOfLengthInBits;
 }
 /* Otherwise we have a normal encoding */
 else {
  (*LengthInBits) = EDBitsToInt (Source, (*OffsetInBits)+1, 7);
  (*OffsetInBits) += 8;
 }

 /* Don't forget that 'length' is in bits! */
 (*LengthInBits) <<= 3;
}
```

4 Information elements

4.1 Bit-fields declarations

Bit-fields are data structures that can be exploited to expand information elements.
Bit-fields have a fixed part, expressed by sequences of bits, and an optional variable part appended to the end of the field.
This module follows the standard ETSI approach when numbering bits. If we have a four-octets set of bits, numbering works as follows (using the `bit-field` declaration):

	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1
octet 1	32	31	30	29	28	27	26	25
octet 2	24	23	22	21	20	19	18	17
octet 3	16	15	14	13	12	11	10	9
octet 4	8	7	6	5	4	3	2	1

If fields are declared with the `zbit-field` keyword, they are numbered starting from 0 (zero) instead of 1. Please, note that octets are still numbered from 1. This complies with some other ETSI/3GPP numberings (like TS 23.040):

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
octet 1	31	30	29	28	27	26	25	24
octet 2	23	22	21	20	19	18	17	16
octet 3	15	14	13	12	11	10	9	8
octet 4	7	6	5	4	3	2	1	0

4.1.1 Bit-fields BNF grammar

BitField ::= ("bit-field" | "zbit-field")<Identifier:typeName> "{"
BF_FixedPartSize [BF_OverallMaxSize]{[BF_Item|BF_SuperItem] ";"} {BF_Remainder}
["son" "of" <Identifier:parentTypeName>] "}" [";"]

BF_FixedPartSize ::= (["fixed" "part"] "size" ":" BF_SizeDescriptor)[";"]

BF_Item ::= BT_Item_Uncond [["shift"] "conditional" CCode->condition]

BF_Item_Uncond ::= ["octet" <Natural:octet>]
("bit" | "bits")(<Natural:from> | <Natural:from> "-" <Natural:to>)[":"] <Identifier:fieldName> FT_FieldType

BF_SuperItem ::= ["octet" <Natural:offsetInBytes>
[["shift"] "conditional" <CCode :condition>]
"{" {BF_Item_Unconditional ";"} "}" ";"

BF_OverallMaxSize ::= (["overall"]("max" | "maximum") "size" ":"
BF_SizeDescriptor)[";"]

BF_Remainder ::= "remainder" ":" <Identifier:fieldName> FT_FieldType
["conditional" CCode->condition] [";"]

BF_SizeDescriptor ::= <Natural:v> ("octet" | "octets" | "byte" | "bytes") | <Natural:v>
("bit" | "bits")

4.1.2 Examples

Example 1 – Basic bit-fields
The following example shows declaration of a simple bit-set which splits an octet into two nibbles called n1 and n2:

```
bit-field BF_Bin1 {
  size: 8 bits;
  bits 1-4: n1 integer;
  bits 5-8: n2 integer;
}
```

Same as above but using the zbit-field numbering:

```
zbit-field BF_Bin1 {
  size: 8 bits;
  bits 0-3: n1 integer;
  bits 4-7: n2 integer;
}
```

Below is shown how to map this bit-field inside an information element:

```
gsm-0407 TESTMSG_IN {
     in signallist TestIn;
     ProtocolDiscriminator = 0101;
     MessageType           = 1X001101;

  AB MyBitfield     M  TV     2 BF_Bin1;
     cond1          M   V     1 integer;
     t5V            M   V     2-10;
}
```

Example 2 – variable sized bit-fields and conditional sub-fields
In this example is shown how to access variable sized structures and conditional fields.

```
bit-field BF_BinRest {
  size: 1 octet;
  max size: 4 octets;
  bits 1-4: n1 integer;
  bits 5-8: n2 integer conditional %{ DATA.n1 <= 0 }%;
  bits 5-8: n3 integer conditional %{ DATA.n1 > 0 }%;
  remainder: nrA conditional %{ DATA.n1 <= 0 }%;
  remainder: nrB conditional %{ DATA.n1 > 0 }%;
}
```

Note that:

- Keyword size declares the size of the fixed part.
- Keywords max size declare the maximum size of the whole information element; in this example, variable part can range from 0 to 3 octets, since 1 octet is always covered by the fixed part.
- Remainder is placed into a binary field called nrA or nrB;
- Conditions are expressed like for conditional TLV fields.

Example 3 – octet based declaration
Instead of bit based declaration, it is optionally allowed to declare bit-fields specifying the octet and the bit in the octet:

```
bit-field BF_Fixed {
  size: 2 octet;
  octet 1 bits 1-8: n0 void default=100;
  octet 2 bits 1-4: n1 octet;
  octet 2 bit  5  : n2 boolean;
  octet 2 bit  6-8: n3 integer;
}
```

Please, note that according to ETSI conventions, bit 9 of octet 2 is bit 1 of octet 1.

Example 4 – bit-field with structured variable part
See description of sequences.

Example 5 – conditional fields accessing parent's fields
Sometimes it is necessary for a child structure to determine a condition accessing parent's fields. In the following example, bit-field "BF_Child" accesses a field named "fld1" of its parent, which is of type "BF_Father":

```
bit-field BF_Father {
  size: 2 octet;
  octet 1 bits 1-8: fld1 integer;
  octet 2 bits 1-8: fld2 BF_Child;
}

bit-field BF_Child son of BF_Father {
  size: 1 octet;
  octet 1 bits 1-8: cnd1 integer
        conditional %{ DATA.parent->fld1 == 0 %};
  octet 1 bits 1-8: cnd2 integer
        conditional %{ DATA.parent->fld1 != 0 %};
}
```

In order to be able to access "parent" pointer, we must declare "BF_Child" as "son of " class "BF_Father". This means that "BF_Child" can only be used as a sub-field of "BF_Father".

Example 6 – grouped conditional fields
It is allowed to specify condition for a grouped set of fields:

```
bit-field BF_BinRest {
  size: 1 octet;
  bits 1-4: n1 integer;
  octet 1 conditional %{ DATA.n1 <= 0 }% {
        bits 5-7: n2a integer;
        bit  8  : n2b boolean;
  };
  bits 5-8: n3 integer conditional %{ DATA.n1 > 0 }%;
}
```

Example 7 – shifting fields
In some GSM specifications, entire sets of bits can be missing from a bit-set according to rules specified. For example, those fields which existence depends on a "`0/1 ext`" bit (see GSM-04.08, chapter 10.5). **ATTENTION: starting from version 1.0.23 of Encodix, a native support for 0/1ext fields has been added. Although still supported, "shift" approach for 0/1ext fields should be considered outdated.**

```
bit-field BF_Shift {
  size: 2 octet;
  max size: 6 octets;
  /* Octet 1 */
  octet 1 {
        bit  8: ext1 boolean;
        bits 1-7: v1 integer;
  };
  /* Octet 1a */
  octet 1 shift conditional %{!DATA.ext1}% {
        bit  8: ext1a boolean;
        bits 1-7: v1a integer;
  };
  /* Octet 1b */
  octet 1 shift conditional %{DATA.ext1a_Present && !DATA.ext1a}%
{
        bit  8: ext1b void default=1;
        bits 1-7: v1b integer;
  };
  octet 2 bits 1-8 v2 integer;
  remainder: nr;
}
```

4.2 0/1-ext bit-fields

Document TS 24.007, chapter 11.2.2.1, describes a way of describing extensible bit-fields by mean of tagged octets (please refer to TS 24.007 for details).
Encodix supports natively this approach.

4.2.1 0/1-ext bit-fields grammar

x01ExtField ::= "`0/1ext-field`" <Identifier:typeName>
["`son`" "`of`" <Identifier:parentTypeName>] "`{`" ZOE_FixedPartSize
[ZOE_OverallMaxSize] {ZOE_Item ";"} {ZOE_Remainder} "`}`" ["`;`"]

ZOE_FixedPartSize ::= (["`fixed`" "`part`"] "`size`" "`:`" ZOE_SizeDescriptor) ["`;`"]

ZOE_OverallMaxSize ::= (["`overall`"] ("`max`" | "`maximum`") "`size`" "`:`" ZOE_SizeDescriptor)
["`;`"]

ZOE_SizeDescriptor ::= <Natural->v> ("`octet`"|"`octets`"|"`byte`"|"`bytes`")| <Natural->v>
("bit"|"bits")

ZOE_Bits ::= ("bit" | "bits") (<Natural:from> | <Natural:from> "`-`" <Natural:to>) ["`:`"]
<Identifier:fieldName> FT_FieldType

ZOE_Item ::= ("`0/1ext`"|"`1ext`") "`octet`" <Natural:octetNo> [<Letter:letter>] ["`*`"]
["`repeat`" <Natural:maxRepetitions>] "`{`" {ZOE_Bits "`;`"} "`}`" ["`;`"]

ZOE_Remainder ::= "`remainder`" "`:`" <Identifier:fieldName> FT_FieldType ["`;`"]

4.2.2 Examples

Example 1 - TS 24.008 10.5.4.16 *High layer compatibility*
Information element 10.5.4.16 *High layer compatibility* is described in TS 24.008 with a table as follows:

8	7	6	5	4	3	2	1	
	High layer compatibility IE							octet 1
Length of high layer compatibility contents								octet 2
1 ext	coding standard		spare			presentation mode of protocol profile		octet 3*
0/1 ext	High layer characteristics identification							octet 4*
1 ext	Extended high layer characteristics identification							octet 4a*

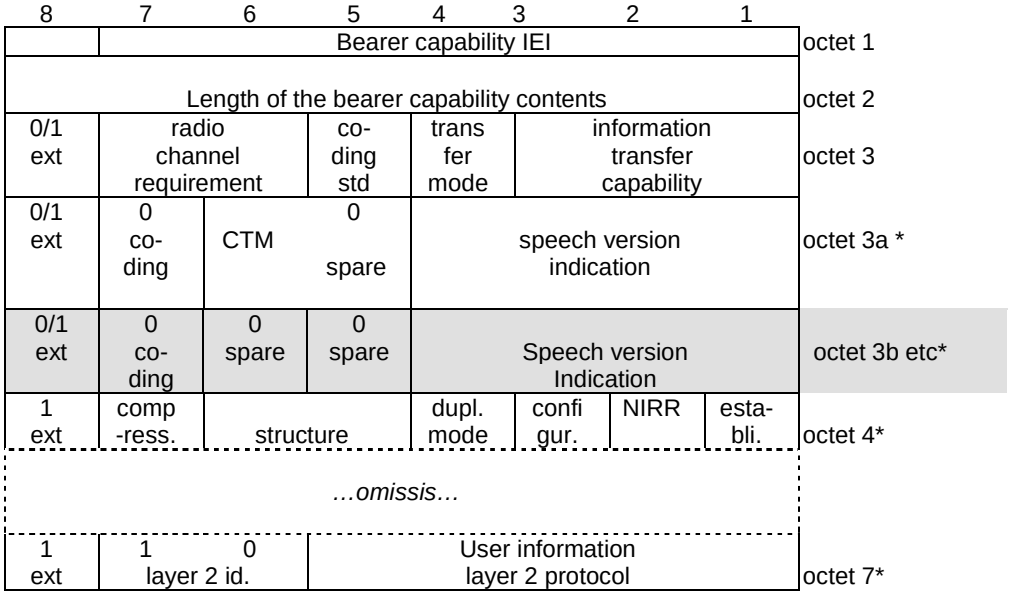
This table can be represented in Encodix as follows:

```
0/1ext-field HighLayerCpbility {
  size: 1 octets;
  max size: 3 octets;
  1ext octet 3* {
        bits 6-7: CodingStandard integer;
        bits 3-5: spare void default=0;
        bits 1-2: PresentationMethod integer;
  };
  0/1ext octet 4* {
        bits 1-7: HighLayerCharacteristicsId integer;
  };
  1ext octet 4a* {
        bits 1-7: ExtHighLayerCharacteristicsId integer;
  };
}
```

Please, note that:

- octets 1 and 2 are not inserted in this field (they are already handled by the "TLV" entry which refers to this type);
- octet number does not matter in this context: Encodix uses octet number only to distinguish a group: "`octet N`" is the leader of a group named "*N*", "`octet N`a" is the next one of that group and so on;
- a star (`*`) following an octet number means that the octet is optional;
- with `ED_01EXT_MULTI_OPTIONAL` set to its defautl value of 0, for optional octets having more than one sub-entry (like `octet 3*` in the example above), only the first entry (`CodingStandard`, in the example above) is marked as "optional"; this means that in generated code, only that entry will have the `_Present` modifier; the remaining fields are read/set according to the situation of the first one;
- with `ED_01EXT_MULTI_OPTIONAL` set to 1, for optional octets having more than one sub-entry (like `octet 3*` in the example above), all the non void entries entry (`CodingStandard` and `PresentationMethod`, in the example above) are marked as "optional"; this means that in generated code, all those entries will have the `_Present` modifier; **however, only the first one (`CodingStandard`) is considered when encoding**.

Example 2 - Expressing repeated entries
The example below is taken from TS 24.008, chapter 10.5.4.5.



The evidenced line (octet 3b) is marked with an "etc" at the end. This means that the octet can be repeated several times; the last entry is marked with a 1 on its 0/1ext bit.
This can be represented in Encodix with the "repeat" keyword:

```
0/1ext octet 1b* repeat 4 {
    bits    1-4:    speechVersion1b integer;
    bits    5-6:    spare1b void default=0;
    bit     7:      coding1b boolean;
};
```

The "`repeat 4`" keyword means that octet 1b can be repeated up to 4 times.
Please note that, given "repeat N":

- Encodix generates an array of 0-N items for each entry under the repeated entry;
- if the star optional indicator (*) is present, Encodix requires 0 to N items;
- if the star optional indicator is missing, Encodix requires 1 to N items.

4.3 Sequence declarations

Sequences are variable arrays of a given type.

4.3.1 Sequence BNF grammar

Sequence ::= "`sequence`" <Identifier:typeName> "`[`" <Natural:maxItems> "`]`"
["`of`"] FT_FieldType ["`size`" FT_Size]
[“`son`” “`of`” <Identifier:parentTypeName>] ["`;`"]

Example 1 – A bit-field with structured variable part
In this example, we show how to declare a bit-field which variable part is made of a repetition of a given structure.

```
bit-field BF_SeqItem {
  size: 2 octets;
  octet 2 bits 1-8: n1 integer;
  octet 1 bits 1-7: n2 integer;
  octet 1 bit  8:   n3 boolean;
}

sequence SQ_Seq1 [4] BF_SeqItem;

bit-field BF_SeqRest {
  fixed part size: 1 octet;
  max size: 9 octets;
  bits 1-8: nx integer;
```

```
   remainder: nr SQ_Seq1;
}
```

In this example, bit-field `BF_SeqRest` has a variable part which is made of a sequence from 0 to 4 instances of `BF_SeqItem`.
SDL code accessing a variable `binSeq` declared of type `BF_SeqRest` could be:

```
binSeq!nx := 241,
binSeq!nr!data(0)!n1 := 190,
binSeq!nr!data(0)!n2 := 77,
binSeq!nr!data(0)!n3 := true,
binSeq!nr!data(1)!n1 := 191,
binSeq!nr!data(1)!n2 := 78,
binSeq!nr!data(1)!n3 := false,
binSeq!nr!items := 2
```

Please note that `binSeq!nr!items` had to be set to the number of items really used in this instance.

Example 2 – A sequence of integers
In this example, we show how to declare a sequence which elements are integers.

```
sequence SQ_Seq1 [6] integer size 4 bits;
```

In this example we want to encode/decode a sequence of at most 6 integer of 4 bits each.

Example 3 – Sequence of conditional fields accessing parent's fields
In this example, we show how to declare a sequence which elements have internally a condition which have to be retrieved from sequence's parent (see also chapter 4.1.2, example 5).

```
# Sequences with conditional parent-related subfields
bit-field BF_SeqRestCnd {
   fixed part size: 1 octet;
   max size: 5 octets;
   bits 1-8: vc integer;
   remainder: vr SQ_CSeq;
}

sequence SQ_CSeq [4] BF_SeqCndItem son of BF_SeqRestCnd;

bit-field BF_SeqCndItem son of SQ_CSeq {
   size: 1 octet;
   bits 1-8 fa integer conditional
        %{DATA.parent->parent->vc == 0}%;
   bits 1-8 fb integer conditional
        %{DATA.parent->parent->vc != 0}%;
}
```

In this example we have `BF_SeqRestCnd` which has a remainder which is a sequence of `SQ_CSeq`. This sequence is made of `BF_SeqCndItem`, which fields are conditioned to `BF_SeqRestCnd`'s field named `vc`.

4.4 Custom information elements

Custom information elements allow complete custom declaration of data structures and encoding/decoding behaviour.
In order to define a custom information element, user must declare:

- a data structure (see the syntax below);
- an encoding function;
- a decoding function.

4.4.1 Custom IE BNF grammar

ST_Field ::= ["`optional`"] <Identifier:fieldName> "`:`" ST_SubType ["`;`"]

ST_RootSubType ::= ST_Structured | ST_Sequence

ST_Sequence ::= "sequence" "[" <Natural:maxItems> "]" ST_SubType

ST_Simple ::= “octet” | “integer” | “boolean” | “binary” ST_Size |
<Identifier:existingType>

ST_Size ::= <Natural:v> ("octet" | "octets") | ("bit" | "bits")

ST_Struct ::= "struct" "{" {ST_Field} "}"

ST_Structured ::= ST_Struct | ST_Union

ST_SubType ::= ST_Structured | ST_Simple | ST_Sequence

ST_TypeDef ::= "dcl-type" <Identifier:typeName>
[“son” “of” <Identifier:parentTypeName>] "{"
"size" (ST_Size) [(".." | "...") (ST_Size)] [";"]
"type" ST_RootSubType [";"]
"encode" [<CCode:EncExtraPars>] [":"] <CCode:CEncode> [";"]
"decode" [<CCode:DecExtraPars>] [":"] <CCode:CDecode> [";"]
"}" [";"]

ST_Union ::= "union" "{" {ST_Field} "}"

4.4.2 Encoding and decoding functions

Both decode and encode functions are generated by Encodix; users must provide their body, i.e. those parts of C able to read the binary buffer and fill the custom data type and vice versa. Encodix provides to the user's code:

- a pointer to an instance of the user declared structure;
- a pointer to a `char` buffer;
- an offset in the buffer in bits, where offset 0 if MSB of first octet;
- in case of decoding, the number of bits available to be decoded.

The encode/decode functions must always return the number of bits decoded/encoded.

4.4.2.1 Encode utilities and variables

When encoding, user's code has the following variables and utilities:

Name	Description
`char* Buffer`	The destination buffer
`const USERTYPE* Source`	A pointer to the source structure, containing the data to be encoded
`CURPOS`	An integer value containing the bit offset where encoding should begin.
`PUSH_INT (val, bits)`	Writes "`val`", which is taken as an unsigned int, to `CURPOS`, encoding it in "`bits`" bits. `CURPOS` is then increased of "`bits`" bits.
`PUSH_BIN (val, bits)`	Writes "`bits`" bits taken from "`val`", a char* buffer, to `CURPOS`, starting from MSB of first octet of "`val`". `CURPOS` is then increased of "`bits`" bits.

`PRESENT (x)`	Can be used to test whether optional field "x" is present or not.
	`if (PRESENT (Source->myOptFld)) {`
	`  // encode also myOptFld`
	`}`

`THIS`	Is a macro defined as an alias for `Source`. It can be disabled by setting `ED_GENERATE_THIS` to 0.

4.4.2.2 Decode utilities and variables

When decoding, user's code has the following variables and utilities:

Name	Description
`const char* Buffer`	The source buffer
`int Length`	Number of bits available for decoding.

USERTYPE* Destin	A pointer to the destination structure, to be filled the data decoded from "Buffer".
CURPOS	An integer value containing the bit offset where decoding should begin.
SHIFT_INT(bits)	Read "bits" bits from "Buffer" starting from "CURPOS" and returns them as an unsigned integer. CURPOS is then increased of "bits" bits.
SHIFT_BIN(dest,bits)	Read "bits" bits from "Buffer" starting from "CURPOS" and writes them inside "dest" starting from MSB of first octet of "dest". CURPOS is then increased of "bits" bits.
PRESENT (x)	Can be used to set optional field "x" presency: `PRESENT (Destin->myOptFld) = ED_TRUE;`
THIS	Is a macro defined as an alias for Source. It can be disabled by setting ED_GENERATE_THIS to 0.

4.4.3 Examples

Example 1 – a custom information element
This is an example of declaration of a 100% custom information element.

```
dcl-type TCustom1 {
    # Define the maximum size of the object
    size 5 octets;

    # Define the data type to be associated with this
    # custom field.
    # In this case, we have a union made of three fields:
    # s, which is an array of 2 (x,y) integer pairs, i,
    # an integer, and j, an octet.
    # Please note that size of this data type does not
    # match size declare above: encoded and decoded data
    # CAN be of different size!
    type union {
      s: sequence [2] struct {
        x: integer;
        y: integer;
      }
      i: integer;
      j: octet;
    }

    encode %{
      switch (Source->Present) {
        case U_c_TCustom1_i: {
          PUSH_INT (0, 8);
          PUSH_INT (Source->u.i, 32);
          break;
        }
        case U_c_TCustom1_s: {
          PUSH_INT (1, 8);
          PUSH_INT (Source->u.s.data [0].x, 8);
          PUSH_INT (Source->u.s.data [0].y, 8);
          PUSH_INT (Source->u.s.data [1].x, 8);
          PUSH_INT (Source->u.s.data [1].y, 8);
          break;
        }
      }
      return 40;
    }%
```

```
    decode %{
      switch (SHIFT_INT (8)) {
        case 0: {
          Destin->Present = U_c_TCustom1_i;
          Destin->u.i = SHIFT_INT (32);
          break;
        }
        default: {
          Destin->Present = U_c_TCustom1_s;
          Destin->u.s.data [0].x = SHIFT_INT (8);
          Destin->u.s.data [0].y = SHIFT_INT (8);
          Destin->u.s.data [1].x = SHIFT_INT (8);
          Destin->u.s.data [1].y = SHIFT_INT (8);
          Destin->u.s.items = 2;
        }
      }
      return 40;
  }%
}
```

4.4.4 Optional parameters

Encode and decode functions for custom types can be extended with extra parameters. These parameters can be used to pass extra information to the functions.
The declaration of the parameters is done by writing the `EncExtraPars` and `DecExtraPars`. This is C-Code text (i.e. surrounded by "`%{`" and "`}%`") and it is inserted as is in the parameters declaration at the end of the standard parameters, right before `ED_C_ENCO_DECL`/ `ED_C_DECO_DECL` macros.
For example:

```
dcl-type TTestType {
  size: 2 octets;
  type union {
        foo: integer;
        boo: integer;
  };
  encode %{, int p1, int p2}%: %{/* encode c-code as usual */}%;
  decode %{, int p3}%: %{/* decode c-code as usual */}%;
};
```

The generated functions will be:

```
long ENCODE_c_TTestType (char* Buffer, long BitOffset,
        c_TTestType* Source, int p1, int p2)
{
  /* encode c-code as usual */
}

long DECODE_c_TTestType (const char* Buffer, long BitOffset,
        c_TTestType* Destin, long Length, int p3)
{
  /* decode c-code as usual */
}
```

When invoking the encode/decode functions, these extra parameters must be supplied. If this invocation is done by Encodix generated code, some special keywords must be used:

```
gsm-0407 TEST_MESSAGE
{
  ProtocolDiscriminator = 0001;
  MessageType           = 00000001;

  MyTestType          M V 2 TTestType enco %{,2,3}% deco %{,4}%;
}
```

Whenever "enco" and "deco" parameters are the same, the call can be made with "encodeco":

```
MyTestType          M V 2 TTestType encodeco %{,2,3}%;
```

4.5 Sub-fields

Sub-fields can be used to expand information elements made of a sequence of not homogeneous variable fields.
This is obtained by declaring a set of “encoding steps”: each encoding step, according to its rules, will consume a variable number of bits and leave the remaining to the following step.

4.5.1 BNF grammar

SubField ::= "sub-field" <Identifier:typeName> "{" {StandardField} "}" [";"]

StandardField ::= ("L3IE" TLV_InformationElement) | CSF_Custom

CSF_Custom ::= "custom" <Identifier:elementName> "{"
"size" [":"] (<Natural:min>) [("to" | ".." | "...") <Natural:max>]
("octet" | "octets" | "bit" | "bits") [";"]
["optional" [";"]]
["encode" <CCode:CEncode> [";"]]
["decode" <CCode:CDecode> [";"]]
FT_FieldType [";"]
"}" [";"]

4.5.2 Examples

Example 1 – Recognize TLV IEs as part of a bigger IE
Sometimes, information elements are further split into sub-fields described as TLV. For example, 08.08 message CIPHER MODE COMPLETE has a TLV field named “Layer 3 Message Contents”. This field contains a sequence of 04.08 information elements. This can be obtained by declaring a sub-field like the following one:

```
sub-field SubFieldTest {
  L3IE  18 foo         O  TV     5 integer;
  L3IE  19 helloWorld  M  TLV    2-5 binary;
}
```

After L3IE follows a normal TLV-like declaration.

Example 2 – Recognize custom sub-fields
This example shows how to recognize custom sub-fields.

```
sub-field SubFieldTest {
  custom   zzz3 {
        size: 2..2 octets;
        decode %{Destin->zzz3 = 1234;}%
        integer;
  };
  L3IE  18 zzz1     O  TV     5 integer;
  L3IE  19 zzz2     M  TLV    2-5 binary;
}
```

4.6 TLVSet subfields

This subfields can be used to decode a buffer containing repeatable TLV entries. This implementation assumes a 1-octet tag and 1-octet length always present.
The difference between this and other TLV implementations is that this one expects any number of TLV entries in any order, even with repeated fields.
Syntax is:

TLVSet ::= "declare" "TLV" "set" <Identifier:TLVSetName> "max" "=" <Value:MaxItems>
["tag-size" "=" Value:TagSizeInBits]
["len-size" "=" Value->LenSizeInBits] "{" {TLVSetItem} "}" [";"]

TLVSetItem ::= "TLV" Identifier:FieldName "tag" "=" ("0x" Hex2:Tag | "default") ["max" "="
<Value:maxLen>] "type" "=" FT_FieldType ";"

For example:

```
declare TLV set TUserDataIE max=20 {
   TLV ConcShort tag=0x00 type=TConcShortMsgs8BitRefNum;
   TLV SpcSms    tag=0x01 max=128 type=binary;
   TLV UnknownIE tag=default type=TUnknownIE;
};
```

The "max=20" declaration tells that we expect at most 20 entries, no matter how mixed.
The "max=128" in SpcSms tells, instead, that that binary field (or whatever variable sized item) is not bigger than 128 octets.
The "tag-size" parameter specifies the size **in bits** of the "TAG" in the TLV entry. Default is 8 (i.e. one octet). The "len-size" parameter is the same as the above but for the "length" field.

4.7 Decoding multiple protocol layers

Sometimes it is necessary to decode multiple protocol layers at once. Typical examples of this need are 08.58 ABis Data Request and Data Indication messages.

4.7.1 Examples

Example 1: 08.58 Abis data request/indication
In 08.58 signalling there are two messages, named Data Request and Data Indication, which have a field named "L3 Information". This field holds a fully encoded 04.08 L3 message.
Normally we receive and send ABis messages from/to an SDL system as usual. But, when we receive a 08.58 Data Indication, instead of sending a DATA_INDICATION signal, we would rather prefer to decode L3Info field and send the decoded 04.08 L3 signal inside the SDL system.
Vice versa, when we send a 04.08 signal from the SDL system, we need to encode it inside a L3Info field of a 08.58 Data Request message.
Unfortunately, both Data Indication and Data Request have two extra parameters, Channel Number and Link Id: we need to manage them inside the SDL system.
Encodix handles this problem by using parameterised message sets and some C coding.
Here we show an example where we declare a message set, ABIS data request/indication, two 04.08 incoming messages and two 04.08 outgoing messages:
declare gsm-0407 message set AbisTransportedSet (TAbisDesc abisDesc);

```
declare gsm-0407 message set AbisTransportedSet (TAbisDesc
abisDesc);

/*-----------------------------------------------
   ABIS DESCRIPTOR
-----------------------------------------------*/
dcl-type TAbisDesc {
   size 0 octets;
   type struct {
         ChNr: integer;
         LnkId: integer;
   }

  encode %{return 0;}%
  decode %{return 0;}%
}
```

```
#------------------
# DATA INDICATION
# 08.58/8.3.2
#------------------
sbs-abis DataIndication {
     in signallist TestIn;
     MessageDiscriminator  = 00000010;
     MessageType           = 00000010;

  01 ChNr                 M TV   2 integer; /* 9.3.1 */
  02 LinkId               M TV   2 integer; /* 9.3.2 */
  0B L3Info               M TLV  >=3;       /* 9.3.11 */
}

#------------------
# DATA REQUEST
# 08.58/8.3.1
#------------------
sbs-abis DataRequest {
     in signallist TestOut;
     MessageDiscriminator  = 00000010;
     MessageType           = 00000001;

  01 ChNr                 M TV   2 integer; /* 9.3.1 */
  02 LinkId               M TV   2 integer; /* 9.3.2 */
  0B L3Info               M TLV  >=3;       /* 9.3.11 */
}

#-------------------------------------------------------
# Messages that can be encoded on a Data Indication
#-------------------------------------------------------
gsm-0408 AbisTransp1_IN {
     in signallist TestIn;
     in message set AbisTransportedSet;
     ProtocolDiscriminator = 0111;
     MessageType           = 11000011;

  aa dummy                 M TV 2 integer;
}

gsm-0408 AbisTransp2_IN {
     in signallist TestIn;
     in message set AbisTransportedSet;
     ProtocolDiscriminator = 0111;
     MessageType           = 11000111;

  aa dummy                 M TV 2 integer;
}
```

```
#-----------------------------------------------------
# Messages that can be encoded on a Data Indication
#-----------------------------------------------------
gsm-0408 AbisTransp1_OUT {
     in signallist TestOut;
     in message set AbisTransportedSet;
     ProtocolDiscriminator = 0011;
     MessageType           = 11000011;

  aa dummy                 M TV 2 integer;
}

gsm-0408 AbisTransp2_OUT {
     in signallist TestOut;
     in message set AbisTransportedSet;
     ProtocolDiscriminator = 0011;
     MessageType           = 11000111;

  aa dummy                 M TV 2 integer;
}
```

The following C function shows how to encode ABIS messages:

```
/*-----------------------------------------------------
   Handles outgoing ABIS messages.
   04.08-on-ABIS messages are part of a message set
   declared as AbisTransportedSet. These messages
   must be encoded on a 08.58 (ABIS) DataRequest
   message.
-----------------------------------------------------*/
int Handle_0408_on_ABIS_Outgoing_Messages (xSignalNode *signal,
    char* buffer, int ReleaseSignal)
{
   int len;
   c_DataRequest dataRequest;
   c_TAbisDesc abisDesc;
   len = Sdl2Buffer_AbisTransportedSet (signal, dataRequest.L3Info.value,
          ReleaseSignal, &abisDesc);

   /* If we don't recognize a signal which is part of AbisTransportedSet
   message set, we give up and return <0 */
   if (len < 0) return len;

   /* Otherwise, let's fill DataRequest's fields */
   /* L3Info.value has been filled previously */
   dataRequest.L3Info.usedBits = len * 8;

   /* Set also channel number and link id */
   dataRequest.ChNr = abisDesc.ChNr;
   dataRequest.LinkId= abisDesc.LnkId;

   /* Now, we encode a data request */
   len = ENCODE_c_DataRequest (buffer, 0L, &dataRequest);

   /* Len is in bits. We should return it in bytes */
   return (len+7)/8;
}
```

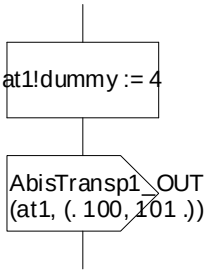
The following C function shows how to decode ABIS messages:

```
/*-------------------------------------------------------------------------
   Returns ED_TRUE if recognized. ED_FALSE otherwise.
   Parameter 'len' is expressed in octets.
   It expects a buffer containing an ABIS message. If the message is a
   "DataIndication", then L3Information is further decoded and a 04.08
   message is sent. Otherwise, the original ABIS message is sent.
-------------------------------------------------------------------------*/
int Decode_And_Send_0408_on_ABIS_Messages (const char* buffer, int len,
        SDL_PId From, SDL_PId To)
{
   /* If we got a DATA INDICATION */
   if (Match_c_DataIndication (buffer, 0)) {
          c_DataIndication dataIndication;
          c_TAbisDesc abisDesc;

          /* Decode data indication */
          DECODE_c_DataIndication (buffer, 0L, &dataIndication, len*8);

          /* Now we can access data indication fields */
          abisDesc.ChNr = dataIndication.ChNr;
          abisDesc.LnkId = dataIndication.LinkId;
          return DecodeAndSend_AbisTransportedSet
                  (dataIndication.L3Info.value,
                   dataIndication.L3Info.usedBits/8, From, To, &abisDesc);
   }
   /* If we got anything different from DATA INDICATION */
   else {
          return DecodeAndSend_ABIS (buffer, len, From, To);
   }
}
```

Here an SDL fragment sending a 04.08 message specifying Channel Number 100 and Link Id 101:



Example 2: decoding a field containing a known complete message
This is the case when a message (for example, a 04.08 message) is fully included on a lower layer message (for example, 08.58 or 08.08).
In this case, there is nothing else to do other than specify the message name as the type of the containing field:

```
#----------------------
# TRANSPORTED MESSAGE
#----------------------
gsm-0408 MY_0408_MESSAGE {
     in signallist TestIn;
     ProtocolDiscriminator = 0101;
     MessageType           = 1X001101;
  …omissis…
}


#----------------------
# TRANSPORTIN MESSAGE
#----------------------
sbs-abis MY_ABIS_MESSAGE {
     in signallist TestIn;
     MessageDiscriminator  = 00000010;
     MessageType           = 00000010;

  01 ChNr                 M TV   2   integer; /* 9.3.1 */
  02 LinkId               M TV   2   integer; /* 9.3.2 */
  0B L3Info               M TLV  >=3 MY_0408_MESSAGE;
}
```

Example 3: decoding a field containing the body of a known complete message
In some standards, like 08.08 for example, there are fields containing the body of a known message. These fields do not contain a complete message but only the data part of it.
For example, see 08.08 Cipher Mode Complete. Here how to support it:

```
gsm-0408 CIPH_MODE_COMPLETE {
     in signallist TestOut;
     ProtocolDiscriminator = 0110;
     MessageType           = 00110010;
  17 mobileId           O   TLV   3-11;
}

#------------------
# INCOMING VERSION
#------------------
gsm-0808 CIPHER_MODE_COMPLETE_IN {
     in signallist TestIn;
     MessageType           = 01010101;

  aa L3Msg                 M TLV 3-30 body of CIPH_MODE_COMPLETE;
}
```

4.8 CSN.1 Modules

CSN.1 module allows declaration of information elements according to CSN.1 format as stated in ETSI GSM 04.07, Annex B. CSN.1 grammar is formal and described in the above document: therefore, we shall not report it here.

4.8.1 BNF Grammar

CSN.1 parts are introduced with a specific keyword described below.

CSN1_Definitions ::= "`csn.1`" [“slave”] "{" {CSN1_Definition}+ "}" [";"]

CSN1_Definition ::= [“`slave`" | “`unslave`" | "`type-only`" | "`incoming`" | "`outgoing`"]
[“`manual-decode`" | "`manual-encode`" | "`manual-encodeco`"]
<04.07 Annex B rule B6>

4.8.2 Notes

There are some things to be underlined:

Names
Names (identifiers) can contain spaces and several other separation characters; in order to guarantee maximum compatibility, names are mangled according to SDL rules, which are the most restrictive:

- all separation characters (i.e. non alphanumeric) are changed into underscores;
- all trailing and leading underscores are removed;
- sequences of more than one underscore are replaced by a single one;
- sequences starting with a number get an underscore added as a prefix.

For example, a name like “`VBS/VGCS options and NLN(SACCH)`” will be changed into “`VBS_VGCS_options_and_NLN_SACCH`”.

ETSI CSN.1 quality
In many cases, CSN.1 declarations provided by ETSI contain syntactical errors. Also, some declarations seem to be far away from specificator intentions, especially regarding definition of lists and sequences.
Check carefully any CSN.1 declaration being copied from official documents!

ETSI CSN.1 extensions
Some ETSI document uses some CSN.1 extensions which are not described in TS 04.07. Encodix supports the following extensions:

`//`	`{a b c //}` is same as `{a {b {c|null} | null} | null};`
`!`	which is like an "or" for unexpected branches;
`=`	which separates send and receive action: `<a> ::= 110 = 001;` means: expect 110 when receiving, but send 001 when sending.

`:=`	Intersection. Used as follows:
	`<Digit> ::= <bit(4) := 0 .. 9>;`
	or
	`<Nine> ::= <bit(4) := 9>;`
	The previous string is accepted only if its numeric value is included in the range specified.
	The intersection check can be disabled by setting the `ED_CSN1_DISABLE_INTERSECTION` TCL macro to 1.

`-`	Exclusion symbol.
	`<TagCh : bit(2) - 11>`
	Accepts everything except for 11.
	`<TagCh : bit(2) – {11|10}>`
	Accepts everything except for 11 or 10.

4.8.3 Data type generation rules

When working with CSN.1, as well as with the other definition models, Encodix has to define data structures able to host the parsed data.

Encodix creates the smallest superset of data element able to contain all the possible configurations of read data.
Data types are chosen according to the following table:

size in bits	type chosen
1	Boolean
2-8	Octet[2]
9-32	Integer (unsigned)
>32	Binary

Other type can be forced by using the custom extension `{type}` (see example 3).
Here some examples:

Example 1

```
<t1> ::= <a: <octet>>;
```

becomes:

```
type t1 {
  i : integer;
}
```

Example 2

```
<t2> ::= 1 <x: <octet>> | 0 <y: <octet>>;
```

becomes :

```
type t2 {
  optional x : integer;
  optional y : integer;
}
```

Example 3

```
<t3> ::= 1 <x : <octet>> | 0 <x : <octet>> <y{binary} : <octet>> ;
```

becomes :

```
type t3 {
  x : integer;
  optional y : binary;
}
```

4.8.4 Rules for structured data

Given the rules to translate a single field into integers, octet, etc., we describe now how structures are handled.

Rule 1 – Explosion of referenced definitions
When a CSN.1 definition references another definition, the parser considers them melted together.
For example:

```
<t1> ::= <x: octet><t2>;
<t2> ::= <x: octet><y: octet>;
```

is considered as being:

```
<t1> ::= <x: octet><x: octet><y: octet>;
```

Rule 2 – Labels
Labels always become a structure field. Encodix generates structure fields only when a label is found and it uses the label name.
For example:

[2] From version 1.0.58. It can be disabled with setting `ED_CSN1_INTEGERS` to 1.

```
<t2> ::= <x: octet>;
```

becomes :

```
struct t2 {
  int x;
};
```

Rule 3 – Nested labels
When labels are nested, corresponding nested structures are generated.
For example:

```
<t2> ::= <a: <x: octet> <y: octet>>;
```

becomes :

```
struct t2 {
  struct {
    int x;
    int y;
  } a;
};
```

Rule 4 – Multiplicity
In the simplest cases, labels have a multiplicity of 1. When a given label can be present in multiple instances, the multiplicity may change.

- Rule 4.a: If multiplicity is “0..1”, Encodix creates an “optional” field;
- Rule 4.b: If multiplicity is “n”, with n > 1, Encodix creates a variable sized array or a fixed sized array if the TCL variable `CSN1_ALLOW_FIXED_SEQUENCES` is set to 1.
- Rule 4.c: if multiplicity is “0..n”, Encodix creates a variable sized array.
- Rule 4.d: if multiplicity is “0..infinite”, Encodix creates a variable sized array which max is `CSN1_INFINITE_ARRAY_SIZE` or a value set in the rule itself (see chapter 4.8.5.1).

The following examples are reported assuming `CSN1_ALLOW_FIXED_SEQUENCES` set to 1.

Example 1 (rule 4.a):

```
<t2> ::= {0 | <x: octet>};
```

becomes :

```
struct t2 {
  int x;
  ED_BOOLEAN x_Present;
};
```

Example 2 (rule 4.b):

```
<t2> ::= <x: octet> * 2;
```

becomes :

```
struct t2 {
  int x [2];
};
```

Example 3 (rule 4.d):

```
<t2> ::= <x: octet> **;
```

becomes :

```
struct t2 {
  struct {
    int data [20];
    int items;
  } x;
};
```

Rule 5 – Multiple labels with the same name
When labels with a given name are reported more than once at the same nesting level, Encodix considers them as being multiple instances of the same label.

Encodix calculates the maximum and minimum number of instances that can be found for a given label and reacts as specified in Rule 4.

Example 1:
In this example, two instances of ‘x’ are declared one after the other.

```
<t2> ::= <x: octet> <x: octet>;
```

is the same as:

```
<t2> ::= <x: octet>*2;
```

and therefore becomes:

```
struct t2 {
  int x [2];
};
```

Example 2:
In this example, two instances of ‘x’ are found alternatively:

```
<t2> ::= {0 <x: octet> | 1 <x: octet>};
```

is the same as:

```
<t2> ::= {0|1} <x: octet>;
```

and therefore becomes:

```
struct t2 {
  int x;
};
```

Example 3:
In this example, two instances of ‘x’ are found alternatively, but only in the second case ‘y’ is present:

```
<t2> ::= {0 <x: octet> | 1 <x: octet> <y: octet>};
```

becomes:

```
struct t2 {
  int x;
  int y;
  ED_BOOLEAN y_Present;
};
```

Example 4:
In this case, the ‘x’ label can be found from 1 to 3 times:

```
<t2> ::= {0 <x: octet> | 1 <x: octet>*3 <y: octet>};
```

becomes:

```
struct t2 {
  struct {
   int data [3];
   int items;
  } x;
  int y;
  ED_BOOLEAN y_Present;
};
```

Example 5:
This is the typical situation where an item is repeated a variable number of times until a terminator string is found, expressed in recursive form:

```
<t2> ::= {0 | 1 <x: octet> <t2>};
```

which is the same of:

```
<t2> ::= {1 <x: octet>}** 0;
```

therefore, it becomes:

```
struct t2 {
  struct {
   int data [20]; /* where 20==CSN1_INFINITE_ARRAY_SIZE */
   int items;
  } x;
};
```

4.8.5 Handling infinite repetitions

Some specifications including infinite repetitions often can not be encoded or decoded without further specifications.
For example, declarations like:

```
<f1> ::= <x: octet> 0**;
```

can be easily decoded (read an octet then zero or more ZERO symbols until the end of the buffer); but what should we do when encoding? How many ZERO symbols should we write? Default behaviour of Encodix is to write none; if different sizes are needed for encoding, “Custom rules” features should be used.

4.8.5.1 Setting the size of the generated array

Infinite sequences can't be translated into infinite arrays, because no hardware couldn't handle them. By default, Encodix generates a structure containing `CSN1_INFINITE_ARRAY_SIZE` items (which defaults to 20).
This value can be changed case by case by adding `[`*number-of-bits*`]` immediately after the repetition expression.
For example:

```
<foo> ::= {1 <m:octet>}** [13] 0; -- Use 13 bits
```

4.8.6 Custom rules

CSN.1 was designed to be formal and to allow complete description of encoding. Unfortunately, some specifications still rely on manual specifications described in comments.
For example, *MS Radio Access capability* information element described in ETSI 24.008, reports:

```
1   <Access capabilities struct> ::=
2       <Length : bit (8)>  -- length in bits of Content and spare bits
3       <Access capabilities : <Content>>
4       <spare bits>** ;    -- expands to the indicated length
5                           -- may be used for future enhancements
```

In this example, at line 4 *spare bits* are declared of infinite size. Actually, the number of bits to be used is specified in a preceding field, *Length*. Explanation of that is in the comment at lines 2 and 4. No computer-based parsers can interpret such text.

To solve this kind of problems, we allow an extended syntax:

```
CSN1_CustomStr    ::= "ENCODE" ":" <CCode:e_code> "DECODE" ":" <CCode:d_code>
                    | "ENCODECO" ":" <CCode:ed_code>
                    | "LIMIT_SIZE" "(" <Natural:Tag> ")"":" <CCode:ed_code>
                    | "RESTORE_SIZE" "(" <Natural:Tag> ")"
                    | "POP_TAGS" "(" <Natural:Tag> ")"
                    | "SAVE" ":" <Natural:PushBits> ["BITS"|"BIT"] "AS" "TAG" "("
                    Natural:PushTag ")"

CSN1_CustomExpr   ::= "ENCODE_EXPR" ":" <CCode:e_code> ("DECODE_EXPR" ":"
                    <CCode:d_code> | "DECODE_INFINITE")
                    | ["ENCODECO_EXPR" ":"] <CCode:ed_code>
                    | "ENCODECO_OPTIONAL" [":"] <CCode:ed_code>
```

4.8.6.1 Custom strings

ENCODE and DECODE statements allow introduction of custom code into the CSN.1 encoding/decoding process. The ENCODE statement contains the C code that is executed during encoding phase, while DECODE is executed during decoding. The ENCODECO shortcut allows writing the same code for encoding and decoding: it is equivalent to writing both ENCODE and DECODE with the same code in them. These statements are often used in conjunction with custom expressions.

Parsing algorithm

Before explaining how C code written inside ENCODE and DECODE statements is executed, we should quickly understand how parsing is done for encoding and decoding.
Encodix uses a “try or backtrack” parsing technique which allows recognition of any CSN.1 string, no matter of its complexity.
For example, let’s assume we have the following binary CSN.1 declaration:

```
<ex1> ::= 0 <x:{1010 | 100}>;
```

If we parse the following binary string:

```
01010
```

the parser will match each element as explained below:

```
01010 => <ex1> ::= 0 {1010 | 100}
01010 => <ex1> ::= 0 {1010 | 100}
01010 => <ex1> ::= 0 {1010 | 100}
01010 => <ex1> ::= 0 {1010 | 100}
01010 => <ex1> ::= 0 {1010 | 100}  ok, string matched.
```

Instead, if we encounter the following string:

```
0100
```

matching would go as follows:

```
0100 => <ex1> ::= 0 {1010 | 100}
0100 => <ex1> ::= 0 {1010 | 100}
0100 => <ex1> ::= 0 {1010 | 100}
0100 => <ex1> ::= 0 {1010 | 100}  error, I was expecting “1”!
Backtrack and try next choice:
0100 => <ex1> ::= 0 {1010 | 100}
0100 => <ex1> ::= 0 {1010 | 100}
0100 => <ex1> ::= 0 {1010 | 100} ok, string matched.
```

Basics of ENCODE/DECODE/ENCODECO statements

If we insert a couple of ENCODECO statements we can see the parser behaviour:

```
<ex1> ::= 0 <x:
   {1 ENCODECO: %{printf ("FIRST  pos=%d\n", CURPOS);}% 010
   |1 ENCODECO: %{printf ("SECOND pos=%d\n", CURPOS);}% 00
}>;
```

Decoding string 01010 we get output:

```
FIRST  pos=2
```

…while decoding string 0100 we get:

```
FIRST  pos=2
SECOND pos=2
```

This demonstrates the backtracking process. Inside embedded C code we can use the following read-only expressions:

Expression	Usage
CURPOS	An integer value containing the number of bits read so far from the beginning of the buffer. See also BUFFER.
BUFFER	A `const char*` variable pointing to the beginning of the buffer being read/written.

4.8.6.2 Custom expressions

Custom expressions can be used when a repetition expression has to be calculated runtime. For example:

```
<Ex2> ::=
        <Length : octet> -- length in octets of Content and fill
        <Content: octet>
        0**; -- expands to the indicated length “val(Length)”
```

This example will generate a C structure like the following:

```
typedef struct _c_Ex2 {
  int Content;
  int Length;
} c_Ex2;
```

In order to expand the final zero-fill to the correct length we need to use a custom expression. This would be done by writing:

```
<Ex2> ::=
        <Length : octet> -- length in octets of Content and fill
        <Content: octet>
        0* ENCODECO_EXPR: %{ <C expression returning the length> }%;
          -- expands to the indicated length “val(Length)”
```

Where we have *<C expression returning the length>* we must write a piece of C code returning the number of bits. Unfortunately we can not simply access the “Length” field: this is because values are stored in the data structure after decoding has terminated. Also, we might be dealing with a recursive field (see related chapter below).
It is possible to solve this issue using the user’s data stack. This data structure is managed by mean of three functions:

```
void PUSH_TAG(int tag, int value);
void SET_TAG(int tag, int value);
int TAG(int tag);
```

By calling PUSH_TAG we can store an integer value in a slot named tag. With TAG we can retrieve the value we stored given its tag. Tags are integer values chosen by us in the range 0 to CSN1_AUTO_TAG_BASE: therefore they allow us to store several distinct values.
The SET_TAG function, instead, changes the value with the given tag without adding a new instance; this can be used to change a value previously added with PUSH_TAG.
In the example below we use an Encodix library function EDBitsToInt which has the following prototype:

```
unsigned EDBitsToInt (const char* Buffer, int Offset, int Bits);
```

This function will read Bits bits from Buffer starting from offset Offset, also expressed in bits, and returns the integer there found. Integers are unsigned and it expects to find values encoded as specified by ETSI.

Example 1: using custom expressions
Here it is the complete example:

```
<ex2> ::=
  <Length : octet> -- length in octets of Content and fill
  ENCODECO:
  %{PUSH_TAG (1, (EDBitsToInt (BUFFER, CURPOS-8, 8)*8+CURPOS));}%
  <Content: octet>
  0* ENCODECO_EXPR: %{TAG(1)-CURPOS}%;
   -- expands to the indicated length "val(Length)"
```

In this example, we calculate the expected end of the fill in bits and we store it in tag #1. Then, we the ENCODECO_EXPR statement, we return the number of bits given subtracting the expected end (TAG(1)) with the position of the first fill 0 (CURPOS).

Example 2: retrieving the maximum value from a list
In this example, taken from 04.60 PSI3 bis message, we have a sequence of CELL_PARAMS_POINTER items; then we have a sequence of "Neighbour parameter set struct" repeated *n* times, where *n* is the maximum value among all the CELL_PARAMS_POINTER items plus one.

```
< Neighbour Cell params 2 struct > ::=
   { 00 -- Message escape
     { 1 < NCP2 Repeat struct >
       < CELL_PARAMS_POINTER : bit (2) >
     } ** 0 --Up to four pointers to the 'Neigbour parameter set
     < Neighbour parameter set :
       < Neighbour parameter set struct >
     > * (1 + max(val(CELL_PARAMS_POINTER)))
   };
```

Here it is the same entry implemented with Encodix:

```
< Neighbour Cell params 2 struct > ::=
     00 -- Message escape
     ENCODECO: %{PUSH_TAG (1,0);}% -- START VALUE IN TAG 1
     { 1 < NCP2 Repeat struct >
         < CELL_PARAMS_POINTER : bit (2) >
         ENCODECO: %{
           if (EDBitsToInt(BUFFER,CURPOS-2,2) > TAG(1)) {
             SET_TAG(1,(EDBitsToInt(BUFFER,CURPOS-2,2)));
           }
         }% -- Save the highest value

     } ** 0 --Up to four pointers to the 'Neigbour parameter set
        < Neighbour parameter set :
          < Neighbour parameter set struct >
        > * ENCODECO_EXPR:%{TAG(1)+1}%;
```

4.8.6.3 SAVE-AS statement

"Save-as" statement is a shortcut for ENCODECO, With the syntax:

```
SAVE: n BITS AS TAG(k)
```

is equivalent to the following code:

```
ENCODECO: %{PUSH_TAG (k, EDBitsToInt (BUFFER, CURPOS-n, n));}%
```

It is useful in cases like:

```
<s> ::= <LENGTH: <bit>(6)>
        <DATA  : <bit> * (len (LENGTH))>;
```

It can be expressed by writing:

```
<s> ::= <LENGTH: <bit>(6)> SAVE: 6 BITS AS TAG(1)
        <DATA  : <bit> * %{TAG(1)+1}%>;
```

4.8.6.4 Using `val()` function

Several CSN.1 declarations make use of a "val(…)" function when expressing repetition counters.
For example:

```
< Reference Frequency struct >::=
   < RFL_NUMBER : bit (4) >
   < Length of RFL contents : bit (4) >
   < RFL contents : octet (val(Length of RFL contents) + 3) >;
```

Starting from Encodix 1.0.28, this concept can be expressed as is, using the syntax shown above.
Encodix generates the same code it would have generated with the following declaration:

```
< Reference Frequency struct >::=
   < RFL_NUMBER : bit (4) >
   < Length of RFL contents : bit (4) > SAVE: 4 BITS AS TAG(n)
   < RFL contents : octet * %{TAG(n)+3}%) >;
```

…where ***n*** is an automatically generated tag number which is never less than `CSN1_AUTO_TAG_BASE`.

4.8.6.5 Using `DECODE_INFINITE`

If we need to use `**` for decoding and `*<expr>` for encoding we may declare:

```
<string>::= <k> * ENCODE_EXPR: %{<expr>}% DECODE_INFINITE;
```

4.8.6.6 Recursive fields and custom expressions

Custom expressions are fully compatible with recursive field declarations. For example, we can declare:

```
<ex3> ::=
  <Length : octet> -- length in octets of Content and fill
  ENCODECO: %{PUSH_TAG (1, (EDBitsToInt (BUFFER, CURPOS-8,
8)*8+CURPOS));}%
  {0 <Content: octet> | 1 <ex3>}
  0* ENCODECO_EXPR: %{TAG(1)-CURPOS}%;
  -- expands to the indicated length "val(Length)"
```

4.8.6.7 The `CSN1_BACKTRACK` C macro

During the custom encoding and decoding operations (i.e. inside `ENCODECO`, `ENCODE` or `DECODE` statements), it is possible to force the parser to fail the match and backtrack.

Example: implementing the '–' or 'exclude' CSN.1 non-standard command
Given the following CSN.1 definition:

```
< foobar > ::=
   < NR_OF_REMAINING_CELLS : { bit (4) - 0000 } >
   |
   0000 0;
```

The meaning of '`bit (4) – 0000`' is "*match any four bits except for 0000*".
This concept can be implemented using the `CSN1_BACKTRACK` command:

```
< foobar > ::=
   < NR_OF_REMAINING_CELLS : bit (4) > 1
   ENCODECO: %{
         if (EDBitsToInt (BUFFER, CURPOS-4, 4) == 0) CSN1_BACKTRACK;
   }%
   |
   0000 0;
```

4.8.7 The `slave` keyword

By specifying the "slave" keyword before a CSN.1 declaration it is possible to exclude generation of C/SDL structures and encoding/decoding functions.
This is useful when a csn.1 string is used only as a substring of another one and not stand-alone.
The entire csn.1 set can be defaulted to slave by specifying `csn.1 slave {`…etc. In that case, some strings can be un-slaved with "`unslave`".

4.8.8 The `incoming/outgoing` keywords

By specifying the "incoming" or the "outgoing" keyword before a CSN.1 declaration it is possible to exclude generation respectively of C/SDL encoding or decoding functions.

4.8.9 The `type-only` keyword

By specifying the `type-only` keyword, only the data types are generated. Please note that in this case no encoding/decoding functions are available at all: therefore, the type can't be used, for example, as an information element. See `manual-encode`/`manual-decode`/`manual-encodeco` below.

4.8.10 The `manual-xxx` keywords

The `manual-encode` keyword forces Encodix to create the type and the prototype for the encoding function, but no body. In this way, the function can be manually implemented in another module.
The keyword `manual-decode` does the same on the decode function, while `manual-encodeco` does it on both.
These keywords have effect only when the other specifiers (like `slave`, `incoming`, `outgoing`, etc.) would have produced an encoding and/or decoding function.

4.8.11 Other examples

Example 1: some CSN.1
This example shows how to insert CSN.1 information elements.

```
csn.1 {
   -- ==========
   --  10.5.1.7
   -- ==========
   <Classmark 3> ::=
         <A5 bits>
         <UCS2 treatment: bit>
         <Extended Measurement Capability: bit>
         { 0 | 1 <Frequency Band List>}
         { 0 | 1 <Mobile System List>}
         { 0 | 1 <UE measurement capability> }
         { 0 | 1 <Multi-slot capability> }
         <spare padding>;
         -- etcetera…
}
gsm-0407 CSN1_A {
     in signallist TestIn;
     in message set InOnly;
     ProtocolDiscriminator = 1000;
     MessageType           = 10001000;
  18 c1                  M  TLV    1-5   Classmark_3;
}
```

Information element C1 of message CSN1_A is expanding its data part according to the CSN.1 specification named “Classmark 3”. Although this, the name used for information element data declaration is the mangled one “Classmark_3”.

4.9 Decoded fields validation

Starting from version 1.0.36, it is possible to declare immediate validation rules for decoding. These rules can be inserted after any type and they are executed just after the value has been decoded.
The main way to add these checks is to add "valid if %{*C-expression*}%" after the data type. *C-expression* is a valid C expression where we have defined the following macros:

- VALUE - is the value of the field itself
- DATA - is a reference to the complete structure.

The written expression must return 0 if the value is not valid, !=0 otherwise.
When a value is invalid, "valid if" solution aborts the decoding and returns an error code <0.

If a more complex check is required, we can use "validator %{*C-code*}%". In this case the given *C-code* is inserted as-is after the field is decoded.

Example 1: checking an integer value of a simple TLV IE
Here it is a simple example:

```
gsm-0407 ExampleMessage {
   in message set Set1, Set2;
   ProtocolDiscriminator = 1000;
   MessageType           = 00000110;

   foo  M  V    1  integer valid if %{VALUE <= 2}%;
}
```

In this example, field "foo" is accepted only if not bigger than 2.

Example 2: checking an integer value of inside an information element
Here it is an example inside a bitfield:

```
bit-field BitField2 {
   fixed part size:3 octet;
   octet 1 bits 1-8 a integer valid if %{VALUE >= 40}%;
   octet 2 bits 1-8 b integer;
   octet 3 bits 1-8 c integer valid if %{VALUE < DATA.b}%;
}
```

In this example, field "a" is accepted only if not smaller than 40. Also, c is valid only if c<b.

5 C output

Encodix generates several C files containing encoding/decoding functions, structures definitions and other support functions.

5.1 Generated structures

Encodix generates a file named `ed_c.h` containing structure definitions.
Each message and each information element will get such a structure.
Basic fields, like integers and booleans, are mapped in a quite intuitive manner into the structures.
Generated structures are named `c_XXX`, where `XXX` is the name of the original data type.

5.1.1 Booleans

Booleans are declared `ED_BOOLEAN`, which is an integer. They should be set to `ED_TRUE` or `ED_FALSE`.

5.1.2 Optional fields

Optional fields can be set to “present” or “missing”. For this reason, if we have a field named `xyz` which is optional, a field named `xyz_Present` will also be generated.

Example 1:

```
typedef struct _c_ATTACH_REQUEST {
  c_MS_network_capability_value_part MSNetworkCapability;
  c_AttachType AttachType;
  c_CipheringKeySequenceNumber GPRSCipheringKeySeqNum;
  int        OldPTMSISignature; /* is optional */
  ED_BOOLEAN OldPTMSISignature_Present;
} c_ATTACH_REQUEST;
```

5.1.3 Binary fields

If a field is declared of type `OCTET_STRING` or left untyped, a binary field is generated.
Binary fields can be of fixed or variable size. If fixed, it is created an array of `ED_BYTE`.
If variable, Encodix generates a struct containing `value`, a fixed array able to contain the biggest string, and `usedBits`, telling how many bits of `value` are to be considered valid.
Please note that bits are used starting by bit 8 of octet 1 of `value` and going on.

Example 1:
This structure contains a field named `mobileId` of fixed size.

```
typedef struct _c_CIPHERING_MODE_COMPLETE {
  ED_BYTE mobileId [9]; /* bits needed 72 */
} c_CIPHERING_MODE_COMPLETE;
```

Example 2:
This structure contains a field named `mobileId` of variable size.

```
typedef struct _c_CIPHERING_MODE_COMPLETE {
  struct {
        ED_BYTE value [9]; /* bits needed 72 */
        int usedBits;
  } mobileId;
} c_CIPHERING_MODE_COMPLETE;
```

5.1.4 Arrays

Arrays are obtained with a standard C array named `data` plus an integer named `items` that tells how many elements of `data` are valid.

Example 1:
This structure contains a field named `mobileId` which is in array of `mobileIdDescriptor`.

```
typedef struct _c_CIPHERING_MODE_COMPLETE {
  struct {
        mobileIdDescriptor data [5];
        int items;
  } mobileId;
} c_CIPHERING_MODE_COMPLETE;
```

5.1.5 Unions

Unions are generated by creating a structure which contains a C union and a selector field named `Present`. The selector field `Present` is an enumeration which values are created by concatenating the following strings:

- literal "U_";
- the generated C type name;
- underscore;
- the field name.

In this way, it is always possible to know which union field is valid.

Example 1:
This union contains two fields: an `octet` named o and an `integer` named `i`.

```
typedef struct _c_TMyUnion {
  enum {
        U_c_TMyUnion_i,
        U_c_TMyUnion_o
  } Present;
  union {
        int i;
        unsigned char o;
  } u;
} c_TMyUnion;
```

It is possible to use this union as demonstrated in the following C code:

```
c_TMyUnion *data;
data = get a valid c_TMyUnion pointer…
switch (data->Present) {
  case U_c_TMyUnion_i: /* here data->i is valid */
  case U_c_TMyUnion_o: /* here data->o is valid */
}
```

5.1.6 Automatic sorting of structures

Structures fields are sorted in decreasing order of their size. This feature can be disabled by setting `ED_DISABLE_STRUCT_SORTING` to 1.
Being platform independent, Encodix can't know the size of each data element: therefore these values can be programmed by the user.

The default values are:

TYPE	SIZE (octets)
*	4
-	0
int	4
unsigned	4
char	1
ED_BYTE	1
ED_BOOLEAN	1

long	4
ED_LONG	4
ED_SHORT	2
ED_OCTET	1

The special type "*" refers to the size of pointers. The special type "-" is used as a default for unlisted types.
These values can be changed or extended by setting the ED_DATA_SIZES variable. Please, refer to the example below for its exact syntax:

```
set ED_DATA_SIZES {
   ED_BOOLEAN  = 2
   int         = 2
   *           = 8
}
```

5.2 Dynamic Data Module

The default philosophy of Encodix is to generate static structures. These structures have no allocation problems like fragmentation or memory leaks, but they must allocate always the maximum amount of potentially required memory. This can be expensive on tight targets and/or with big messages.
The optional "Dynamic Data Module" allows the generation of data structures where variable items are dynamically allocated.
The Dynamic Data Module is not compatible with the SDL module: when generating SDL code, the Dynamic Data Module is disabled. However, support for Dynamic Data under SDL will be developed in the future.

5.2.1 Activating the dynamic generation

Once you have obtained the "Dynamic Data Module" license, you may activate the dynamic generation by setting the TCL variable ED_DYNAMIC_DEFAULT to 1 in your localconfig.tcl configuration file:

```
set ED_DYNAMIC_DEFAULT 1
```

5.2.2 Functions used for dynamic allocation

Encodix does every allocation by calling a macro named "EDAlloc"; it performs every deallocation by calling "EDFree". These macros are by default set to malloc and free, but they can be overridden by setting them inside ed_user.h.

5.2.3 General rules for dynamic types

It is absolutely mandatory to initialize and free the dynamic structures before and after use. Otherwise, crashes due to wild pointers and memory leaks are guaranteed.
Therefore, the dynamic objects must be handled has follows:

```
c_MyMessage myMessage;
INIT_c_MyMessage (&myMessage);
```

…use it!

```
FREE_c_MyMessage (&myMessage);
```

In the example above, an entire c_MyMessage is allocated on the stack; only its variable parts (like optional elements, union fields, variable arrays, etc.) will be allocated dynamically on the heap.
To obtain an object completely allocated on the heap:

```
c_MyMessage* myMessage;
myMessage = (c_MyMessage*)EDAlloc (sizeof (c_MyMessage));
INIT_c_MyMessage (myMessage);
```

…use it!

```
FREE_c_MyMessage (myMessage);
```

```
EDFree (myMessage);
```

5.2.4 Dynamic structures

When the dynamic model is active, structures (`struct`) are generated implementing statically the mandatory elements and dynamically the optional ones. Small types (like integers, chars, etc.) are always implemented statically even if optional.
For example:

```
gsm-0407 MyMessage {
  ProtocolDiscriminator = 0101;
  MessageType           = 0100x011;

  01 fldA   M  TV 7 TSub;
  02 fldB   O  TV 7 TSub;
  03 fldC   O  TV 3 integer;
}
```

whould generate:

```
typedef struct _c_MyMessage {
   c_TSub * fldB;
   ED_SHORT fldC;
   ED_BOOLEAN fldB_Present;
   ED_BOOLEAN fldC_Present;
   c_TSub fldA;
} c_MyMessage;
```

For every optional field, Encodix generates a function or a macro (depending on its complexity) that manages its presence. These macro/functions are named with the template:

```
<ctype>_<field> (<ctype>* sp, int present)
```

For example, to set `fldB` present, we would call:

```
SETPRESENT_c_MyMessage_fldB (&myMessage, ED_TRUE);
```

These macros will allocate or free the memory where needed according to the `present` flag; if an already present field is set to present or a missing one is set to missing, the macro will do nothing.

Please note that such macros/functions are always generated, no matter if the related field is not dynamic.

Notice: the SETPRESENT_xxx_yyy function must be called BEFORE accessing the "yyy" field.

5.2.5 Dynamic unions

Unions are generating using dynamic data for complex items and static data for simple fields (like integers, chars, etc.). For example:

```
dcl-type TCustom1 {
   size 5 octets;
   type union {
         a: integer;
         b: TSub;
   }

  encode %{}%  decode %{}%
}
```

generates:

```
typedef enum {
   U_c_TCustom1_NONE,
   U_c_TCustom1_a,
   U_c_TCustom1_b
} TPRESENT_c_TCustom1;
```

```
typedef struct _c_TCustom1 {
        TPRESENT_c_TCustom1 Present;
        union {
              ED_LONG a;
              c_TSub *b;
        } u;
} c_TCustom1;
```

As for the structures, the currently present field in the unions must be set by calling specific functions/macros. Encodix generates the following macros/functions:

```
GLOB_SETPRESENT_<ctype> (<ctype>* sp, TPRESENT_<ctype> which);
SETPRESENT_<ctype>_<field> (<ctype>* sp);
```

The two forms are equivalent (the latter is a macro call to the first one). The `which` parameter is an element of the enumeration `TPRESENT_<ctype>` generated for the union <*ctype*>.
For example:

```
GLOB_SETPRESENT_c_TCustom1 (&myVar, U_c_TCustom1_b);
```

or, as an alternative:

```
SETPRESENT_c_TCustom1_b (&myVar);
```

Notice: the SETPRESENT_xxx_yyy function must be called BEFORE accessing the "yyy" field.

5.2.6 Dynamic sequences

Sequences (i.e. C arrays) can be either fixed or variable; *fixed* sequences are those which have a fixed number of items; *variable* sequences are those which have a variable number of items, from 0 up to a given maximum number.

5.2.6.1 Fixed sequences

Fixed sequences get their items allocated at once when the hosting sequence is allocated.
Example:

```
gsm-0407 Msg1 {
  ProtocolDiscriminator = 0111;
  MessageType           = 11100011;

  01 fldA   M  TV  5   MyFixedSeq;
  03 fldC   O  TV  5   MyFixedSeq;
}

sequence MyFixedSeq [fixed 4] octet;
```

This generates:

```
typedef struct _c_MyFixedSeq {
  ED_OCTET data [4];
} c_MyFixedSeq;

typedef struct _c_Msg1 {
  c_MyFixedSeq * fldC;
  ED_BOOLEAN fldC_Present;
  c_MyFixedSeq fldA;
} c_Msg1;
```

Usage example:

```
c_Msg1 a;
INIT_c_Msg1 (&a);

a.fldA.data [0] = 1;
a.fldA.data [1] = 2;
a.fldA.data [2] = 3;
```

```
a.fldA.data [3] = 4;

SETPRESENT_c_Msg1_fldC (&a, E);
a.fldC->data [0] = 10;
a.fldC->data [1] = 20;
a.fldC->data [2] = 30;
a.fldC->data [3] = 40;

FREE_c_Msg1 (&a);
```

5.2.6.2 Variable sequences

Variable sequences allocate their items on demand. To manage such sequences, Encodix generates the following functionsmacros:

```
SETITEMS_<ctype> (<ctype>* sp, int n);
ADDITEMS_<ctype> (<ctype>* sp, int n);
```

`SETITEMS` allocates n items in the structure; if n is greater than the number of currently allocated items, the array is enlarged; the existing items are not deleted nor changed; if n is smaller than the number of allocated items, the exceeding items are deleted.
`ADDITEMS` adds n items to the currently allocated ones.
The internal `items` sequence variable is automatically managed by the macros above and doesn't need to be set manually anymore.

5.2.7 Dynamic binary variables

Binary variables are designed to contain a raw sequence of bits. As for sequences, binary variables can be fixed or variable.

5.2.7.1 Fixed binary data

Fixed binary data is created as a fixed array of ED_BYTE elements; it is created of the minimum number of ED_BYTEs needed to contain the requested number of bits.
Example:

```
gsm-0407 MsgBin1 {
  ProtocolDiscriminator = 0111;
  MessageType           = 11001001;

  07 fldA   M  TV    9 binary;
  08 fldB   O  TV    9 binary;
}
```

In the example above, we have `fldA`, allocated statically, and `fldB`, allocated dynamically.
Generated structure is:

```
typedef struct _c_MsgBin1 {
   ED_BYTE fldA [8];
   ED_BYTE* fldB;
   ED_BOOLEAN fldB_Present;
} c_MsgBin1;
```

Please, note that in this case, although `fldB` is allocated dynamically, it is used exactly as `fldA`:

```
c_MsgBin1 a;
INIT_c_MsgBin1 (&a);
memset (a.fldA, 123, 8);

SETPRESENT_c_MsgBin1_fldB (&a, ED_TRUE);
memset (a.fldB, 45, 8);

FREE_c_MsgBin1 (&a);
```

5.2.7.2 Variable binary data

Variable binary data is made of a structure containing the `ED_BYTE` data pointer and a integer `usedBits` variable. Its allocation is managed through a macro generated by Encodix:

```
ALLOC_<ctype>_<fld> (<ctype>* sp, int bits);
```

Please note that every call to this macro erases the existing buffer. The `usedBits` variable is set to `bits`.
Example:

```
gsm-0407 MsgBin2 {
  ProtocolDiscriminator = 0111;
  MessageType           = 11011001;

  09 fldA   M  TLV 2-6 binary;
  0A fldB   O  TLV 2-6 binary;
}
```

It generates the following data structures:

```
typedef struct _c_MsgBin2_fldA {
  ED_BYTE* value;
  int usedBits;
} c_MsgBin2_fldA;

typedef struct _c_MsgBin2_fldB {
  ED_BYTE* value;
  int usedBits;
} c_MsgBin2_fldB;

typedef struct _c_MsgBin2 {
  c_MsgBin2_fldB* fldB;
  ED_BOOLEAN fldB_Present;
  c_MsgBin2_fldA fldA;
} c_MsgBin2;
```

Usage example:

```
c_MsgBin2 a;
INIT_c_MsgBin2 (&a);
ALLOC_c_MsgBin2_fldA (&a.fldA, 16);
memset (a.fldA.value, 123, 2);

SETPRESENT_c_MsgBin2_fldB (&a, ED_TRUE);
ALLOC_c_MsgBin2_fldB (a.fldB, 24);
memset (a.fldB->value, 45, 3);

FREE_c_MsgBin2 (&a);
```

5.3 Generated functions

Encodix generates several functions that can be invoked by the user.

5.3.1 Encoding functions

Given a generated data structure `c_XXX`, a function named `ENCODE_c_XXX` is created. It has the following prototype:

```
long ENCODE_c_XXX (char* Buffer, long BitOffset, const c_XXX*
Source);
```

This function will encode the structure `Source` in the buffer `Buffer`, allocated by the user, starting from offset `BitOffset` (in bits).
It returns the number of bits used or <0 in case of error.

5.3.2 Decoding functions

Given a generated data structure c_XXX, a function named DECODE_c_XXX is created. It has the following prototype:

```
long DECODE_c_XXX (const char* Buffer, long BitOffset, c_XXX*
Destin, long Length);
```

This function will decode the binary data found in the buffer Buffer at offset BitOffset (in bits) using no more than Length bits from the source. Decoded data is stored in Destin.
It returns the number of bits used or <0 in case of error.

5.3.3 Set-decoding and encoding functions

Each message set has a recognize-and-decode function. This function is named:

```
int SetDecode_xxx (const char* buffer, Txxx_Data* data, int bitLen);
```

where "*xxx*" is the name of the set. It returns the number of bits used or <0 in case of error.
The value "bitLen" is the length of the data in "buffer" expressed in bits.
To maintain compatibility with Encodix 1.0.33 and older, there is another form of set-decoding function:

```
int Decode_xxx (const char* buffer, int len, Txxx_Data* data);
```

This function returns ED_TRUE if the message was recognized and decoded correctly. Otherwise, it returns ED_FALSE. Also, "len" is expressed in octets.

5.3.3.1 SetEncode function

Encodix is optionally able to generate the SetEncode function. This feature is activated by setting the following TCL variable:

```
set ED_GENERATE_SET_ENCODE 1
```

This will generate a SetEncode function which resembles the related SetDecode:

```
int SetEncode_xxx (char* buf, int bitOfs, const Txxx_Data* data);
```

where 'buf' is the destination buffer, 'bitOfs' is the offset in bits where the data has to be encoded, (0=encodes at the beginning of 'buf'), and 'data' is the set object to be encoded.

5.3.3.2 Set structure

The structure here named T*xxx*_Data, where *xxx* is the name of the set, is a structure which contains an union of all the messages structures of the set.
For example, let's consider a set named MySet, containing three messages: Message1, Message2 and Message3.
Encodix generates:

```
typedef enum {
  ID_MySet_Unrecognized = 0,
  ID_MySet_Message1,
  ID_MySet_Message2,
  ID_MySet_Message3
} TMySet_Type;

typedef struct {
  TMySet_Type Type;
  int ProtocolDiscriminator;
  int MessageType;
  union {
        c_Message1 fld_c_Message1;
        c_Message2 fld_c_Message2;
        c_Message3 fld_c_Message3;
        char Dummy; /* Avoids empty unions! */
  } Data;
} TMySet_Data;
```

If we need to decode an unknown message, we can issue:

```
TMySet_Data MySet;
INIT_MySet_Data (&MySet);
ret = SetDecode_MySet (buffer, & MySet, lengthOfBuffer);
```

Case 1: a valid "Message2"
If we have a valid "Message2" we get:

- `ret >= 0`;
- `ret` == `lengthOfBuffer`, unless for some reason the decoded buffer is shorter;
- `MySet.Type == ID_MySet_Message2`;
- `MySet.Data.fld_c_Message2` is a `c_Message2` structure, containing all the fields decoded from the message;
- `MySet.ProtocolDiscriminator` and `MySet.MessageType` contain the protocol discriminator and message type of the decoded message;

Case 2: a unrecognized message
If the message in the buffer is not recognized, we get:

- `ret == ED_UNKNOWN_MESSAGE`;
- `MySet.Type == ID_MySet_Unrecognized`;
- `MySet.ProtocolDiscriminator` and `MySet.MessageType` contain the protocol discriminator and message type of the message found.

5.3.4 Match functions

For each message `MMM`, a function named `Match_c_MMM` is generated. It has the following prototype:

```
int Match_c_MMM (const char* Buffer, long BitOffset);
```

It returns `ED_TRUE` if in `Buffer`, at offset `BitOffset` in bits a message `MMM` is found.

5.4 Decoding badly formed messages

Starting from version 1.0.34, Encodix provides an extensive support for badly formed messages.
Encodix is able to detect only syntactical errors, i.e. those errors related to the message binary formatting; semantic errors, like bad transaction identifiers etc., are handled by the application.

5.4.1 General rules

Encodix generates code which tries to intercept any possible formatting error. If an error is detected, generated code calls a macro which should handle it.
Default implementation for most of those macros is to abort decoding and to return a specific error value.
However, the user can redefine these macros, for example in "`ed_user.h`" and define different behaviours.

5.4.2 Message too short

If the decoded message is too short, the `ED_HANDLE_MESSAGE_TOO_SHORT_ERROR` macro is invoked. Default behaviour is to return `ED_MESSAGE_TOO_SHORT`.

5.4.3 TLV field with length out of range

If the decoded message contains a TLV field which length is out of the declared length range, the `ED_HANDLE_IE_SIZE_ERROR` macro is invoked. Default behaviour is to return `ED_IE_SIZE_ERROR`.

5.4.4 Unknown or unforeseen message type

Encodix detects unknown message types; it is up to the application to decide whether a given message is unforeseen or not.
The set-decoding function returns `ED_UNKNOWN_MESSAGE` if the message is unknown.

5.4.5 Unknown and unforeseen IEs

When decoding a message, Encodix might encounter an unexpected information element. Encodix can realize that it met an unexpected IE only if the expected IE is taggeg (i.e. it has an IEI).
The reasons why we can get an unexpected IE are multiple:

- the IE was send erroneously by the sender;
- it is an out-of-sequence IE;
- it is a repeated IE.
- it is an IE which belongs to a newer specification.

Unfortunately, there is no way for Encodix to detect the exact case: several different philosophies could be taken.
When it detects an unexpected IEI, Encodix operates as follows:

- if it is waiting for an optional or conditional tagged IE, it considers the IE "missing" and leaves the found IE for the next step;
- if it is waiting for a mandatory IE, it calls a macro named `ED_EXPECT_MANDATORY_IE`;
- if it is waiting for an optional or a conditional IE, it calls a macro named `ED_EXPECT_OPTIONAL_IE`.

The default implementation of `ED_EXPECT_MANDATORY_IE` is:

- if the IEI is not the expected one, the entire IE is discarded and the procedure repeated;
- if the message ends, it returns `ED_MESSAGE_TOO_SHORT`;
- if the expected IEI is found, it goes on.

5.4.5.1 Skipping the unknown IEs

Encodix default implementation tries to skip the unknown IE. But there is a problem: paradoxically, unknown IE can be skipped only if they are known. In order to skip an IE we must know its type (see 24.007, chapter 11.2.1.1); in fact:

- we can't detect untagged IE (V or LV types);
- we must use the embedded length if the IE is TLV;
- we must know the length if the IE is TV.

For this reason, Encodix builds a list of known information elements summarizing it from the input source and uses it when trying to skip an unexpected IE; if the IE is not on the list, it returns `ED_UNKNOWN_IEI`.
Encodix users are invited to work on the `ED_EXPECT_MANDATORY_IE` macro to tailor it on their needs.

5.4.6 Missing mandatory IEs

Encodix default implementation keeps on searching the mandatory IE, skipping all the IE found in the meantime (see 5.4.5), until either it is found or the message ends[3]. Therefore, if the searched mandatory IE is not found, it will return `ED_MESSAGE_TOO_SHORT`.

5.4.7 Conditional IE errors

Conditional tagged IEs (i.e. TV or TLV IEs) are managed as optional information elements; at the end of the message, their condition is checked. This allows to check conditions that depends on information written in IE following the conditional one.
In case of failure of the condition, it returns `ED_MISSING_REQUIRED_CONDITIONAL_IE` or `ED_FOUND_UNEXPECTED_CONDITIONAL_IE`.
Please note that if we receive an unexpected IE before a tagged conditional or optional one, the conditional or optional IE is marked as "absent". This default implementation can be reprogrammed by the user (see 5.4.5),

5.4.8 Defined return values

Default Encodix implementation of decode function returns the following values:

Return value	Description

[3] Or some other error occurs, like `ED_UNKNOWN_IEI`.

>=0	The message is valid; returned value is the number of bits consumed.
ED_SYNTAX_ERROR	The message has a syntax error inside an IE; this error is a catch-all for cases not covered by other return values.
ED_UNKNOWN_MESSAGE	The message is unknown (i.e. message type, protocol discriminator, etc. are unknown).
ED_UNKNOWN_IEI	Encodix found an information element which IEI is unknown.
ED_MESSAGE_TOO_SHORT	The message was too short.
ED_MISSING_REQUIRED_CONDITIONAL_IE	A conditional IE condition was true but the the IE was missing.
ED_FOUND_UNEXPECTED_CONDITIONAL_IE	A conditional IE condition was false but the the IE was found.
ED_MANDATORY_IE_SYNTAX_ERROR	Syntax error decoding a mandatory IE
ED_CONDITIONAL_IE_SYNTAX_ERROR	Syntax error decoding a conditional IE
ED_OPTIONAL_IE_SYNTAX_ERROR	Syntax error decoding an optional IE
ED_FIELD_OUT_OF_RANGE	A field is out of range (returned by sub fields decoders when using "valid if" validation)
ED_PACKEDBUFFER_ERROR	Error occurred during a packet buffer operation.
ED_IE_SIZE_ERROR	Received a "L" information element which embedded length is out of the declared range.

6 SDL Output

In order to support SDL, Encodix generates the items described below. Some of these items can be enabled, disabled or configured by editing the generation options. Please see chapter 11.2 to know more about this.
The generated items are:

- an SDL-PR file named `ed_pr.pr`;
- a *recognition* c functions set named `ed_pr_recog.h/.c`;
- a *management* c functions set named `ed_pr_c.h/.c`;

6.1 The generated SDL-PR file

The generated SDL-PR file contains the following items:

- a `newtype` declaration per each message;
- several support "`newtype`", mainly to describe nested structures and arrays;
- an `encode` and a `decode` operator for each `newtype` supporting it;
- a "`signal`" declaration per each message; each signal is named after the original message name and transports a structure which contains all the message fields.
- some "`signallist`", following what has been declared in the message description file.

6.1.1 SDL `NEWTYPE` conventions

Generated SDL sorts follow some conventions here described.

Structures
Structures are rendered by using standard SDL structures.

Optional fields
Optional fields are obtained by specifying the `optional` keyword in SDL. Please refer to Telelogic SDL Suite documentation to know how to access presence information.

Unions
Unions are implemented through the SDL `CHOICE` type declaration.

Arrays
Arrays are implemented creating a structure which contains a field named `data`, which is an SDL `CArray`, and a field named `items`, which contains the number of items actually used. Remember, when setting arrays, to set also the `items` value.

6.1.2 Encode/decode operators

For each message `newtype`, `encode` and `decode` operators may be generated. They have the following declaration:

```
NEWTYPE myType
  ...
OPERATORS
  encode: myType -> Octet_String;
  decode: Octet_String, myType -> Boolean;
ENDNEWTYPE
```

The `encode` operator takes the source structure and returns an `octet_string` containing the encoded message.
The `decode` operator takes the source `octet_string`, the destination structure and returns TRUE if the decoding has been successful.

7 Visual Basic integration

Encodix generated code can be integrated with Microsoft™ Visual Basic.
Generated C code can be compiled into a DLL and then invoked from Visual Basic.
When Visual Basic integration is active, Encodix generates:

- a `.def` file containing the exports for the DLL;
- a `.bas` file containing the data structures describing messages and the `DECLARE` statements needed to import the DLL functions.

7.1 Activation of Visual Basic generator

Visual Basic generator is activated by setting the configuration variable `generateVB` to 1 (see chapter 10 for information about configuration variables):

```
set generateVB  1
set ED_VB_DLL "C:\\MyFiles\\whatever\\MyEncodix.DLL"
```

Variable `ED_VB_DLL` must be set to the pathname of the generated DLL. See chapter 11.2.9 for information about the meaning of Visual Basic configuration variables.

7.2 Usage of the Visual Basic integration

The Visual Basic project must include the external .BAS module generated by Encodix. Its default name is `ed_vb.bas`.
Visual Basic interface exposes the same functions and data structures that are generated for C: therefore, same naming conventions and rules are retained.

7.2.1 Decoding

Decoding is done by invoking the `DECODE_xxx` function, where xxx is the name of the message to be decoded.

```
Dim PktDlAssMsg As VB_Packet_Downlink_Assignment_message_content
Dim Message As String
Message = …fill it with the binary data to be decoded…
DecodedLenInBits = _
        DECODE_c_Packet_Downlink_Assignment_message_content _
        (Message, 0, PktDlAssMsg, Length)
```

Now decoded data can be accessed by reading the structure `PktDlAssMsg`.

7.2.2 Encoding

Encoding is done by invoking the `ENCODE_xxx` function, where xxx is the name of the message to be encoded.

```
Dim HostBuffer As String
Dim PktDlAssMsg As VB_Packet_Downlink_Assignment_message_content
PktDlAssMsg = …fill the fields with the data to be encoded…

' Alloc enough octets to host the message
HostBuffer = String(128, 0)

EncodedLenInBits = _
   ENCODE_c_Packet_Downlink_Assignment_message_content _
  (HostBuffer, 0, PktDlAssMsg)
```

At the end, `HostBuffer` contains the encoded data.

8 Multi-file

Encodix allows declaration of multiple source files that can be processed separately. While with "INCLUDE" statements multiple files are merged together, with "multi-file" feature they are processed separately, as they were different projects. In this way, build execution time can be dramatically reduced, since only changed sources are rebuilt.
If compared to the single-file generation, multi-file has the following differencies:

- when working with multi-files, we have several separate sources: one of them must be declared as the "MASTER", and the others as "GROUP"s;
- every source begins with a "DEFINITION" section; this section holds all the message-set and signal-list declarations; all the "DEFINITION" parts must be identical;
- no signal-list or message-set can be declared outside the "DEFINITION" part;
- instead of a single .pr file for SDL, separate .prm files will be generated; they will be merged by an utility named "prmerge".

Suggestion: since DEFINITION part must be identical for every source set, we suggest to write it in a stand-alone file and to include it at the beginning of all the others.

8.1 Declaration of source files

Here it is an example of a definition file (named, for example, "definitions.src"):

```
DEFINITIONS
declare groups grp1, grp2;

declare signallist TestIn IN, TestOut OUT;
declare sbs-abis message set abisMsg;
declare gsm-0407 message set Gulp;

declare gsm-0408 outgoing message set Set1 (lapdLL ll);
declare gsm-0408 incoming message set Set2 (lapdLL ll);

declare gsm-0407 message set AbisTransportedSet (TAbisDesc
abisDesc);
declare gsm-0808 message set nanoMsg;
declare gsm-0407 incoming message set InOnly;
declare gsm-0407 outgoing message set OutOnly;
```

Note that at the beginning of the definition file it is necessary to declare all the groups we are going to implement.
Example of a master file:

```
`INCLUDE definitions.src`

MASTER
gsm-0407 TESTMSG_IN {
     in signallist TestIn;
     ProtocolDiscriminator = 0101;
     MessageType           = 1X001101;

          t1Va                          M   V     1/2;
  etcetera…
```

What follows "MASTER" is a normal source file for Encodix.

Example of a group file:

```
`INCLUDE definitions.src`

GROUP grp1

gsm-0407 GULP_IN {
     in signallist TestIn;
  etcetera…
```

8.2 External data types

It is allowed to define a data type (for example, a bit field) in a group and then use it in another group.
In order to use an external data type, this has to be declared “external” where it is used.
Group declaration order is significant:

- master file can use every type;
- group declared first can “see” types declared in groups declared next;
- this means that it is not allowed to “cross-define” data in different groups.
- CSN.1 declarations can not be split: please see “Splitting CSN.1” below.

Example 1: declaring a bit-field in an external group
I declare MyData in group “grp1”:

```
GROUP grp1

bit-field MyData {
  size 2 octets;
  …etcetera…
```

Then I use it in group “grp2”:

```
GROUP grp2

external MyData;

gsm-0407 GULP_IN {
   …etcetera…
   0E myField    M   TLV  2-4 MyData;
  …etcetera…
```

8.2.1 Splitting CSN.1

It is not allowed to split a CSN.1 declaration in different groups; this should not be a problem: most CSN.1 declarations are short.
However, different CSN.1 declarations can be made in different groups: basic CSN.1 library can be duplicated, since it does not generate anything by itself. Generally speaking, any CSN.1 string without label (see rule A1 on ETSI 04.07 or 24.007 documents) can be declared once in different groups at the same time.

8.3 Batch file

In order to work with multiple files, execution batch file must be modified.

- all master/group files must be processed separately;
- PRMERGE utility must be invoked in order to merge the .PRM files in a single .PR.

Example of a .BAT file:

```
cd ..\..\gsm-l3
..\bin\codegenix gsm-l3.tcl ../testing/test2/regrtest.src ../testing/test2/gen
                config=../testing/test2/localconfig.tcl
..\bin\codegenix gsm-l3.tcl ../testing/test2/grp1.src ../testing/test2/gen
                config=../testing/test2/localconfig.tcl
..\bin\codegenix gsm-l3.tcl ../testing/test2/grp2.src ../testing/test2/gen
                config=../testing/test2/localconfig.tcl
cd ..\testing\test2
..\..\bin\prmerge gen\ed_pr.pr gen\*.prm
```

9 Dump module

This optional module provides a C code generator which generates “dump” functions.
These functions can be used to “dump” data structures (like messages or information elements) in readable ASCII format.
This feature can be used, for example:

- To deploy Encodix as a traffic logger
- To help debug an application
- To check if the Encodix source is correct
- Etc.

9.1 Principles

Default behaviour of Encodix is to generate a C data structure per each logical element defined inside an Encodix source.
For example, if I define a message named “ATTACH_REQUEST”, Encodix will generate a C structure called c_ATTACH_REQUEST whose fields are the information elements of the message. Also, CSN.1 declarations generate their own data structures, and so on.
First thing that Encodix Dump module does is to generate a function per each data structure which is able to read an instance of it and to print it on a file.
For example, in the example above it will generate a function named “Dump_c_ATTACH_REQUEST “ which accepts a pointer to a “c_ATTACH_REQUEST“ and prints its contents on a file, following any nested data structure and printing everything at the maximum detail.

Another central concept of Encodix is the *message set*. You can take some messages which are all coming from the same direction and formatted in the same way and group them in a message set.
For example, you can group ATTACH_REQUEST, ATTACH_COMPLETE, DETACH_REQUEST, etc. in a single message set named L3Uplink, because they are all uplink messages coming from the same source type.
Given a message set, Encodix generates a data structure which is the union of all the data structures of the messages composing the message set. For example, it would create a structure named TL3Uplink_Data, which contains a discriminator field (which tells which message is contained) and a union of c_ATTACH_REQUEST, c_ATTACH_COMPLETE, c_DETACH_REQUEST, etc.
Encodix generates a function named “Decode_<*message set*>” which takes a binary stream, decodes it, understand which message it contains and fills the given T<message set>_Data structure with the required information.
Encodix Dump module generates a function named “Dump_<*message set*>”, which takes a T<message set>_Data and writes it on a file.

9.1.1 Dump abstraction

Encodix Dump module is not tied to a specific way of writing information.
First of all, it allows to specify any ASCII stream: although possible, it is not mandatory to use stdio’s FILEs. This generic stream is called **TEDOStream**.
Second, it writes the elementary types (integers, octets, structures, arrays, unions, etc.) calling some functions which are provided by us in a .c source file; these functions can be changed at will to get the wanted layout.
Third, some items can have an inline C-code specification that is used to print them; in this way it is possible, for example, to print mnemonic values instead of numeric ones.

9.2 Using Dump module

9.2.1 Generating, compiling and linking

Note: before using Dump module be sure your license allows it!
Dump module is activated by setting the startup variable “generateDump” to 1 (see “Extended configuration” chapter).
Once activated, dump source files are generated automatically. Their default name is `ed_c_dump.h/c`.
If the Dump module is used, some library files have to be compiled and linked in the program:

`ed_dump.c`	Contains some basic parts and it has always to be included
`ed_dump_file.c`	Contains an implementation of TEDOStream called TEDOStreamFile, which works on a regular file.
`ed_dump_val.c`	Contains the functions which dump each type of data. It can be redefined by the user to get a different layout.

9.2.2 Updating the source files to introduce custom dumps

If we write a specification like:

```
bit-field AttachType {
  size: 4 bits;
  bit    4: FORx boolean;
  bits 1-3: TypeOfAttach integer;
}
```

what we obtain is that FORx field will be written as “TRUE” or “FALSE” and TypeOfAttach will be written as a plain decimal number.
If we want to write a mnemonic value for “TypeOfAttach” we may use the “enum” extension, highlighted in bold:

```
bit-field AttachType {
  size: 4 bits;
  bit    4: FORx boolean;
  bits 1-3: TypeOfAttach integer enum {
                    0 = "zero"
                    1 = "one"
                    2 = "two"
                    default = "other"
                };
}
```

Finally, we might need a completely special layout; in this case, we can specify directly C code:

```
bit-field AttachType {
  size: 4 bits;
  bit    4: FORx boolean;
  bits 1-3: TypeOfAttach integer dump %{
              char Buffer [18];
              if (DATA == 3) {DUMPSTR ("Type of Attach is three");}
              else {
                    sprintf (Buffer, "Value=%d", (int)DATA);
                    DUMPSTR (Buffer);
              }
        }%;
}
```

The names “DATA” and “DUMPSTR” are macros automatically generated by Encodix.

10 Access module

Encodix standard C module is designed to produce a static C structure meant to be manually accessed by the programmer. This suites the needs of a standard application, where each field is treated by a specific piece of code which is implementing the protocol.
Other applications, instead, have different needs. Let's think of the case of a "message editor". This application runs on a host machine and allows the user to compose test messages by setting all the values manually: the application then encodes such messages into their binary form on a file; vice versa, the same application must be able to take a binary message and to present it in readable form to the user, for examination or further editing.
The static structures produced by the standard Encodix C module would make the development of our "message editor" quite annoying. In fact it would force us to manually writing specific code for each specific message, information element and down to each subfield, just because we have to access the structures. This approach, applied to our "message editor" example, could partially nullify the advantage of having the automatic code generation of Encodix, because it would force us to maintain both Encodix source files and the editor code.
Here is where the Access module comes in.

10.1 Basic concepts

The Access module generates C++ code that implements two concepts: *type description* and *data description*.
We define the following Encodix source file as an example:

```
gsm-0407 LOCATION_UPDATING_REQUEST {
  ProtocolDiscriminator = 0101;
  MessageType           = xx001000;

   LocationUpdatingType   M V   1/2 integer;
   CiphKeySequenceNum     M V   1/2 integer;
   LocationAreaId         M V   5   LocationAreaId;
   MobileStationClassmark M V   1   MobileStationClassmark1;
   MobileId               M LV  2-9 MobileId;
33 MobStatClsMrkForUmts   O TLV 5   MobileStationClassmark2;
};

bit-field MobileStationClassmark1 {
   size: 1 octets;
   bit 8:    spare void default = 0;
   bits 6-7: RevisionLevel integer;
   bit  5:   EsInd boolean;
   bit  4:   A5_1  boolean;
   bit 1-3:  RfPowerCapability integer;
}
```

Type description
Type description is a tree of C++ objects which describes the data types. For example, a type descriptor would say something like:

> *A LOCATION UPDATING REQUEST message is a structure containing the following fields: LocationUpdatingType, CiphKeySequenceNum, LocationAreaId, MobileStationClassmark, …etcetera; the LocationUpdatingType field contains a 4-bit integer; the MobileStationClassmark, instead, is itself a structure: it contains RevisionLevel, a 2-bit integer, EsInd, a boolean… etcetera.*

The type descriptor above can be traversed by a program that has no knowledge of the "LOCATION UPDATING REQUEST"; our hypothetical "message editor" could use that description to dynamically create an appropriate dialog box for editing such messages.
If the description of the " LOCATION UPDATING REQUEST" message changes, we just need to re-run Encodix and recompile the "message editor" application.

Data description
Once we have dynamically designed our dialog box, the user can fill out all the fields. We still have a problem: ho do we transfer all this data to Encodix so it can encode it? And how do we access the data once Encodix has decoded a binary message?

This is addressed by the "data description". The "data description" is a tree of C++ objects describing the data itself; for example, a data descriptor would say something like:

> *We have a LOCATION UPDATING REQUEST message; the LocationUpdatingType field is 2, while the CiphKeySequenceNum is 7; the field EsInd of MobileStationClassmark is set to FALSE, while… etcetera.*

Once again, the data description can be traversed by out program, which can use it to fill the dialog box fields.

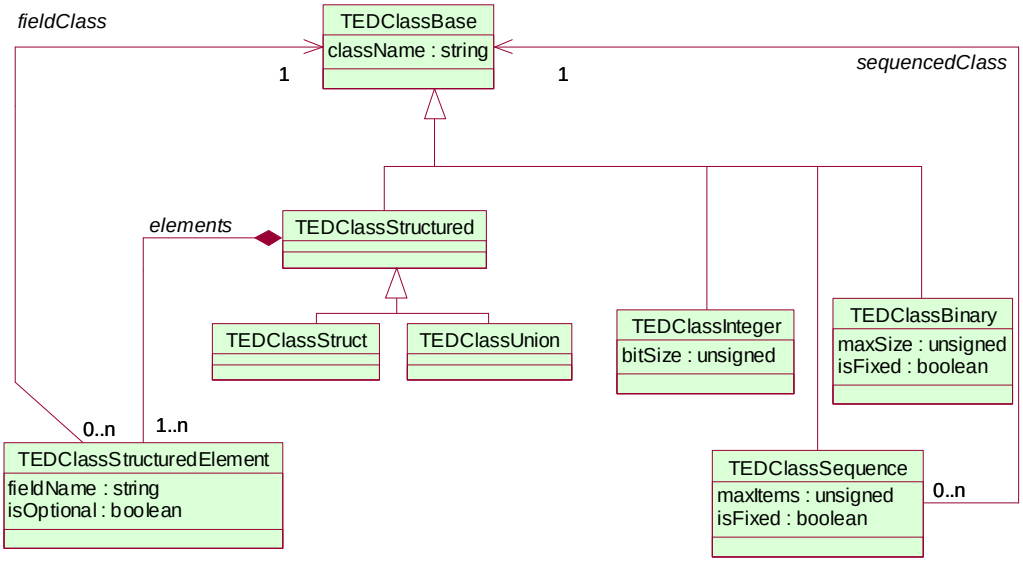
10.2 The Access classes

Access module uses the classes defined in two include files distributed together with Encodix:

- `ed_access_class` for type description classes;
- `ed_access_data` for data description classes.

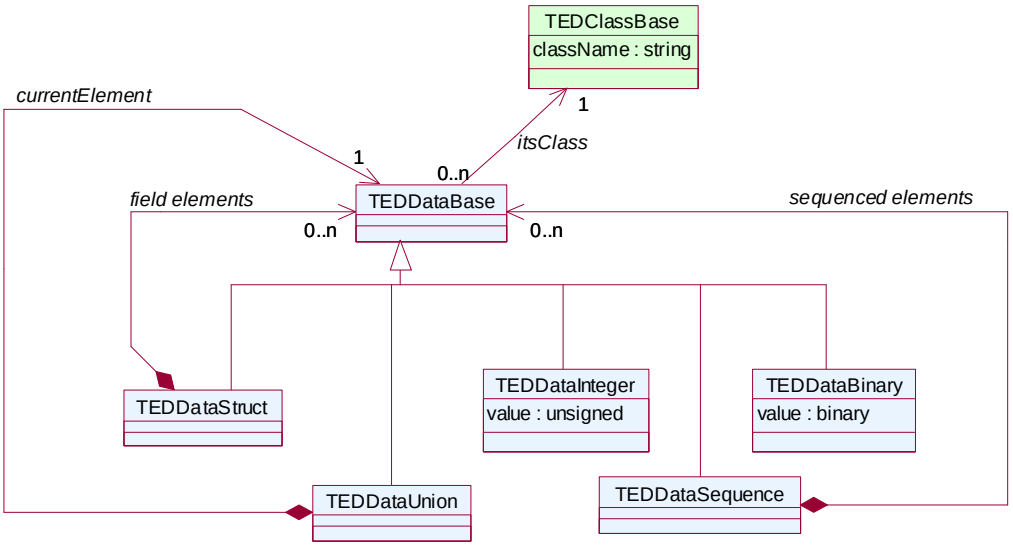
10.2.1 Type description metamodel

The diagram below summarizes the class hierarchy of the type description classes.



10.2.2 Data description metamodel

The diagram below summarizes the class hierarchy of the data description classes.



10.2.3 Usage

<TBD>

11 Extended configuration

Encodix generation can be customised to match most common users' needs. This customisation is done by setting some TCL variables in a user-written TCL file. The pathname of this file is passed to Encodix through the `config=` option described in chapter 2.2.

11.1 TCL short summary

A TCL file is a plain ASCII file. For extended configuration we need only a few features of TCL.

Comments
Comments in TCL are prefixed by # and the end at end of line.

Setting a variable
A TCL variable is set with the following syntax:

```
set <varname> <value>
```

For example:

```
set ED_SDL_PR_GENERATE_ENCODE_OPERATOR 1
```

Expressing complex right-values
Strings containing spaces or symbols should be quoted; TCL accepts escaped characters like C:

```
set MY_TEXT "This is a text\n use a 21\" monitor"
```

11.2 Configuration variables

11.2.1 Generation of SDL support files

SDL support files are generated when the following variable is set:

```
set generateSDL 1
```

11.2.2 Generation of SDL encode/decode operators

Encodix can generate SDL encode and decode operators for each generated `NEWTYPE`.
To activate generation of `encode` operator, set:

```
set ED_SDL_PR_GENERATE_ENCODE_OPERATOR 1
```

To activate generation of `decode` operator, set:

```
set ED_SDL_PR_GENERATE_DECODE_OPERATOR 1
```

To make these operators compatible with TAU SDL Suite 4.2 new operator model, declare:

```
set ED_SDL_PR_COMPACT_OPERATORS 1
```

11.2.3 SDL newtypes prefix

SDL types are named after the name declared in the message description file. Users can specify a string which is prefixed to each SDL type name. Default prefix is "T_":

```
set ED_SDL_PR_PREFIX "T_"
```

11.2.4 Passing extra parameters to encode/decode functions

Sometimes it is necessary to have extra information distributed to encoding and decoding functions.
This can be obtained by using the some TCL variables; in the following example we demonstrate how to pass an extra TData parameter named connInfo to all encoding functions:

```
set ED_C_ENCO_DECL ", TData* connInfo"
set ED_C_ENCO_CALL ", connInfo"
set ED_C_ENCO_SDL  ", TData /*#REF*/"
```

Macro ED_C_ENCO_DECL is appended to all encoding function declarations:

```
long ENCODE_c_X (char* Buffer, long BitOffset, const c_X* Source, TData* connInfo);
```

Macro ED_C_ENCO_CALL is appended to all encode function invocations:

```
Ret = ENCODE_c_X (Buffer, BitOffset, Source, connInfo);
```

Macro ED_C_ENCO_SDL is appended to all encode SDL operators:

```
NEWTYPE T_X /*#NAME 'TSDL_T_X' */ STRUCT
OPERATORS
  encode: T_X/*#REF*/, TData /*#REF*/ -> Octet_String; /*#OP(B)*/
ENDNEWTYPE
```

Same thing applies for decoding functions: macros are the same, except for "ENCO" which should be changed to "DECO".

11.2.5 Doing extra actions before sending an SDL signal

A piece of C code can be inserted just before the invocation to xGetSignal in the DecodeAndSend functions.
This is done by setting the TCL variable named ED_PRE_XGETSIGNAL.

11.2.6 Generation of "tight" C structures

If TCL variable ED_C_TIGHT is set to 1 (default is 1 from version 1.0.61), then C code generator will use ED_OCTET and ED_SHORT where 8-bit and 16-bit numeric values are found.

11.2.7 Defining generated file names

In the following table $OUTPUT_DIR is set to the path chosen by the user when invoking Encodix.

NAME	DEFAULT	DESCRIPTION
ED_C_OUTPUT_DUMP_C	$OUTPUT_DIR/ed_c_dump.c	Dump module .c file
ED_C_OUTPUT_DUMP_H	$OUTPUT_DIR/ed_c_dump.h	Dump module .h file
ED_C_OUTPUT_H	$OUTPUT_DIR/ed_c.h	File containing c structures, output of ED-C
ED_CFGFILE	$OUTPUT_DIR/Encodix_configs.h	File containing global inter-file includes
ED_CSN1_C	$OUTPUT_DIR/CSN1DataTypes.c	Files containing CSN.1 parser
ED_CSN1_H	$OUTPUT_DIR/CSN1DataTypes.h	Files containing CSN.1 parser
ED_CSN1_PREFIX	*Empty string*	Prefix added to the global static structures in generated CSN.1 code. Used to allow linking of multiple CSN.1 generated files. The string GROUPNAME is replaced with the name of the group, if working in master/group mode.
ED_GROUP_CFGFILE	$OUTPUT_DIR/GROUPNAME_configs.h	File containing group-related inter-file includes
ED_KNOWNIE_C	$OUTPUT_DIR/ed_c_known_ie.c	Known IEs .c file

ED_KNOWNIE_DB	$OUTPUT_DIR/ed_c_known_ie.db	Known IEs .db file
ED_KNOWNIE_H	$OUTPUT_DIR/ed_c_known_ie.h	Known IEs .h file
ED_RECOG_C	$OUTPUT_DIR/ed_c_recog.c	File containing recognizer code
ED_RECOG_H	$OUTPUT_DIR/ed_c_recog.h	File containing recognizer code
ED_SDL_PR	$OUTPUT_DIR/ed_pr.pr	File containing SDL-PR
ED_SDL_PR_C_C	$OUTPUT_DIR/ed_pr_c.c	File containing SDL-PR conversion tools
ED_SDL_PR_C_H	$OUTPUT_DIR/ed_pr_c.h	File containing SDL-PR conversion tools
ED_SDL_RECOG_C	$OUTPUT_DIR/ed_pr_recog.c	Files containing SDL recognizer code
ED_SDL_RECOG_H	$OUTPUT_DIR/ed_pr_recog.h	Files containing SDL recognizer code
ED_SUBFIELDS_C	$OUTPUT_DIR/SubFields.c	File containing sub-fields implementation.
ED_SUBFIELDS_H	$OUTPUT_DIR/SubFields.h	File containing sub-fields implementation.
ED_USER_H	ed_user.h	User defined include file (to add any user required definition, etc.)
ED_USERDEFINED_C	$OUTPUT_DIR/UserDefinedDataTypes.c	Files containing user defined data types
ED_USERDEFINED_H	$OUTPUT_DIR/UserDefinedDataTypes.h	Files containing user defined data types

The file extensions are chosen according to the values below:

ED_C_EXT	.c	Extension for C source files
ED_H_EXT	.h	Extension for C include files
ED_CPP_EXT	.cpp	Extension for C++ source files
ED_HPP_EXT	.h	Extension for C++ include files

Library source files names in generated code can be changed by setting the following variables:

NAME	DEFAULT
ED_LIBH_SCTTYPES	scttypes.h
ED_LIBH_ALLIFC	all_ifc.h
ED_LIBH_BITENCODE	bitencode.h
ED_LIBH_CSN1LIB	csn1lib.h
ED_LIBH_ED_DATA	ed_data.h
ED_LIBH_ED_LIB	ed_lib.h
ED_LIBH_ED_LIB_SDL	ed_lib_sdl.h
ED_LIBH_ED_TLV	ed_tlv.h
ED_LIBH_ED_DYNAMIC	ed_dynamic.h[4]
ED_LIBH_ED_ACCESS	ed_access.h
ED_LIBH_ED_DUMP	ed_dump.h
ED_LIBH_ED_DUMP_VAL	ed_dump_val.h

11.2.8 Other variables

LICENSE_FILE	Path to the license file. Default path is: <*EncodixBase*>/license/Encodix.lic
ED_CHECK_STRUCT_DUPES	When generating data structures, checks for duplicated fields. If working on big files, this option may be disabled (duplications will be caught by the C compiler). Default is 1.

[4] Note: variable ED_LIBH_ED_DYNAMIC, prior to version 1.0.66 was called ED_LIBH_DYNAMIC_H. The name has been changed to make it consistent with the names of the other variables. However, to allow backwards compatibility, the old name is still accepted.

USER_C_HEADER	User defined C header. It is inserted on every generated C file.
generateDump	If set to 1, module Dump is activated. This may require a specific license.
ED_MAX_CSN1_FILE_SIZE	If set to a value >0, defines the maximum size allowed for CSN.1 generated files in bytes. Every time that size is exceeded, the function being written is completed and a new file is started.
ED_DISABLE_MESSAGE_SIGNATURE	If set to 1, disables the generation of recognizer's ProtocolDiscriminator and MessageType fields introduced in version 1.0.35
ED_GENERATE_EXPORT_CALL	Default to 1. If set to 0, the ED_EXPORT_CALL macro is never used. That macro is useful only when “export” are needed. The macro is harmless, but it can disabled to avoid confusing some editors (regarding automatic parameter expansion).
ED_ENHANCED_CSN1	Default is 1. If set to 0, activates the old, unoptimized CSN.1 code generator. If the old code generator is activated, the C macro CSN1_OLD_ENCODER must be globally defined.
CSN1_INFINITE_ARRAY_SIZE	Default is 20; default size for infinite repetitions in CSN.1.
ED_CSN1_INTEGERS	Default is 0. If set to 1, Encodix generates integers instead of octets also when bit sequences are 8 bit long or less.
ED_01EXT_MULTI_OPTIONAL	If set to 1, the items of an optional 0/1ext octet will all be marked as "optional".
ED_GENERATE_ED_CONST	Default is 1. If set to 0, Encodix doesn't generate the ED_CONST macro.
ED_DYNAMIC_DEFAULT	Default is 0. If set to 1 activates the Dynamic Data Module.
ED_DISABLE_STRUCT_SORTING	If set, disables the automatic sorting of C structures.
ED_GENERATE_THIS	Default is 1. If set to 0, disables the generation of the THIS macro.
ED_KNOWNL3IES	Default is “KnownL3IEs". Used to change the name of that global variable.
ED_DISABLE_ENCODIX_VERSION	Default is 0. If set to 1, Encodix version is not reported in generated files.
CSN1_ALLOW_FIXED_SEQUENCES	Default is 0. If set to 1, Encodix generates fixed structures when CSN.1 sequences have a fixed size.
ED_CONVERT_UNSPECIFIED_CONDITIONS_TO_OPTIONAL	Default is 1. Conditional tagged IEI are now generated as "optional" if no condition is specified. This feature can be disabled by setting this variable to 0.
ED_GENERATE_SET_ENCODE	Default is 0. If set to 1, the “SetEncode” functions are generated.
ED_CSN1_DISABLE_INTERSECTION	If set to 1, it disables the range check when the CSN.1 “:= n..m” intersection values are specified. This means that the “:=…” part is logically removed.
ED_C_ACCESS_POSTFIX	Default is the empty string. String that is appended to the Access module generated functions and variables (namely InitializeAccessClass,

	`DeleteAccessClass` and `EDArrayOfClasses`) to avoid naming conflicts.
`ED_C_TYPE_PREFIX`	Default is “c_”. String that is prefixed to the generated C types.

11.2.9 Visual Basic configuration variables

The following variables are used when generating Visual Basic files:

NAME	DEFAULT	DESCRIPTION
`ED_VB`	`$OUTPUT_DIR/ed_vb.bas`	File containing Visual Basic interface
`ED_VB_PREFIX`	`VB_`	Prefix for Visual Basic types
`ED_VB_DEF`	`$OUTPUT_DIR/ed_vb.def`	Generated .def file for Visual Basic

`ED_VB_DEF_HEADER`	`LIBRARY Encodix`
	`DESCRIPTION "Encodix DLL"`
	`EXPORTS`
		Header of the .def file. It should be redefined setting the desired DLL name and description. This can be used also to manually add other exports if needed.

`ED_VB_DLL`	*Undefined*	Path to the compiled DLL. It is used in the `DECLARE` statements to find the DLL.
`ED_VB_DECL_FOOTER`		Text appended to the generated .BAS file. It can be used, for example, to add other `DECLARE` statements if needed.

11.2.10 Generated C types configuration variables

The following variables can be used to change the names of the generated C basic types:

TCL Variable	Default
`ED_NAME_BOOLEAN`	`ED_BOOLEAN`
`ED_NAME_FALSE`	`ED_FALSE`
`ED_NAME_TRUE`	`ED_TRUE`
`ED_NAME_BYTE`	`ED_BYTE`
`ED_NAME_LONG`	`ED_LONG`
`ED_NAME_SHORT`	`ED_SHORT`
`ED_NAME_OCTET`	`ED_OCTET`

12 License

In order to run, Encodix must be provided with a license. This license is delivered by Dafocus as a small text file.
Default license name and location is: <*Encodix root*>`/license/Encodix.lic`.
This path can be changed by setting the configuration variable `LICENSE_FILE` (see chapter 9 to know how to set configuration variables).

12.1 License types

Encodix can be delivered with one of the following license types:

- demo
- time-locked
- complete

Demo License
Demo license is automatically activated if no license file is found.
This license allows usage of all Encodix features only on authorized source files. Authorized source files can be slightly modified and Encodix still accepts them.

Time-locked
A time-locked license expires when the due date is reached. When this license is applied, Encodix talks with `objlicd`, the Dafocus license server daemon. See chapter 12.2 to know how to install and use `objlicd`.

Complete
Complete license allows unlimited usage of all the authorized features. It does not need a connection to `objlicd`, the Dafocus license server daemon.

12.2 License server daemon

The Dafocus license server daemon (`objlicd`) is a TCP/IP socket server which waits for clients to connect.

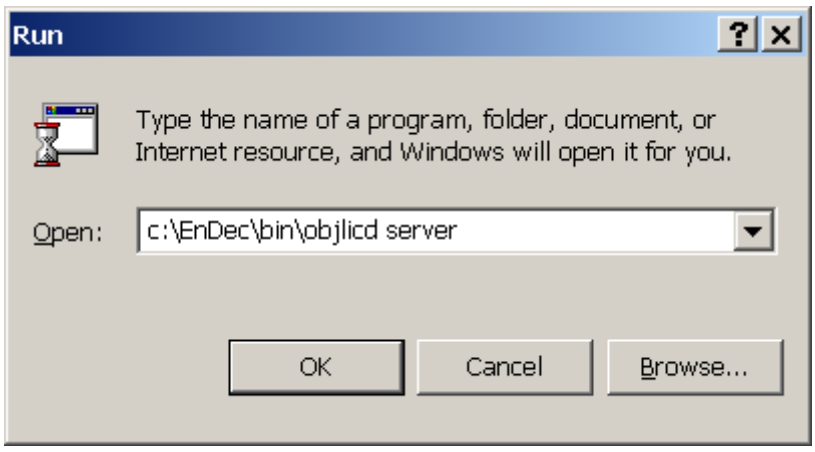
12.2.1 Installing `objlicd`

The `objlicd` daemon can be run standalone or a Windows NT service.
The `objlicd` executable is located under the `bin` directory below the Encodix directory root.

Running standalone
To run `objlicd` standalone execute:

```
<Encodix root>\bin\oblicd server
```

For example, assuming Encodix has been installed under `C:\Encodix`:



To stop the server, hit `CTRL-C` in `objlicd` window.

Running as a service
`objlicd` can be registered as a service. In this way, it will be automatically started at system startup and it will run nicely in the background.
To register it as a service:

```
<Encodix root>\bin\oblicd register
```

To un-register (i.e. remove it from the services list):

```
<Encodix root>\bin\oblicd unregister
```

Accessing a license server on a remote machine
The license server can be executed on a network machine. By default, Encodix tries to connect to `localhost`; if you need to connect to a different server, edit the license file and change `localhost` with your server hostname and port (see the examples below):

`localhost:1234`	connect locally on port 1234
`10.0.0.22:9876`	connect to 10.0.0.22 on port 9876
`myserver.dafocus.com`	connect to that host on default port
`server1:1821`	connect to server1 on port 1821

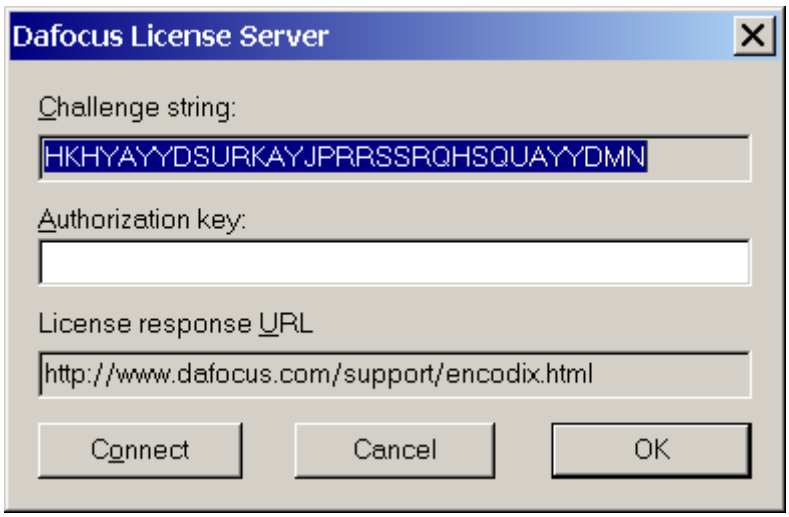
12.2.2 Options

The license server, when executed with the `register` or `server` commands, accepts the following options:

`-p`*<port>*	Server waits on TCP/IP port *<port>* instead of default port 6429.
`-l`*<path>*	Log is written on a file named *<path>* instead of default file, which is located under the same directory of `objlicd.exe` and it is named `objlicd.log`.

12.2.3 Challenge/response method

The Dafocus license server acts as a *cache* for authorizations which are indeed released by Dafocus's online web server.
The first time `objlicd` receives a request from Encodix, it pops up a dialog like the following:



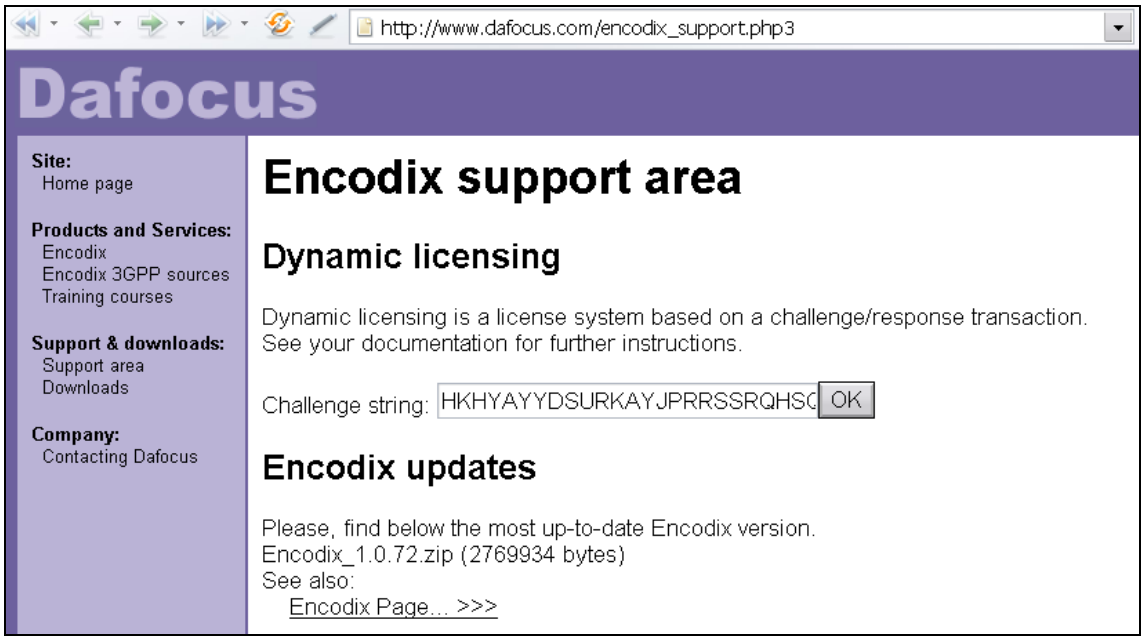
WARNING: sometimes this dialog box appears below some other windows; if Encodix is waiting for a response, look for this dialog box in your toolbar.

At this point, the user should connect to the following Internet URL:

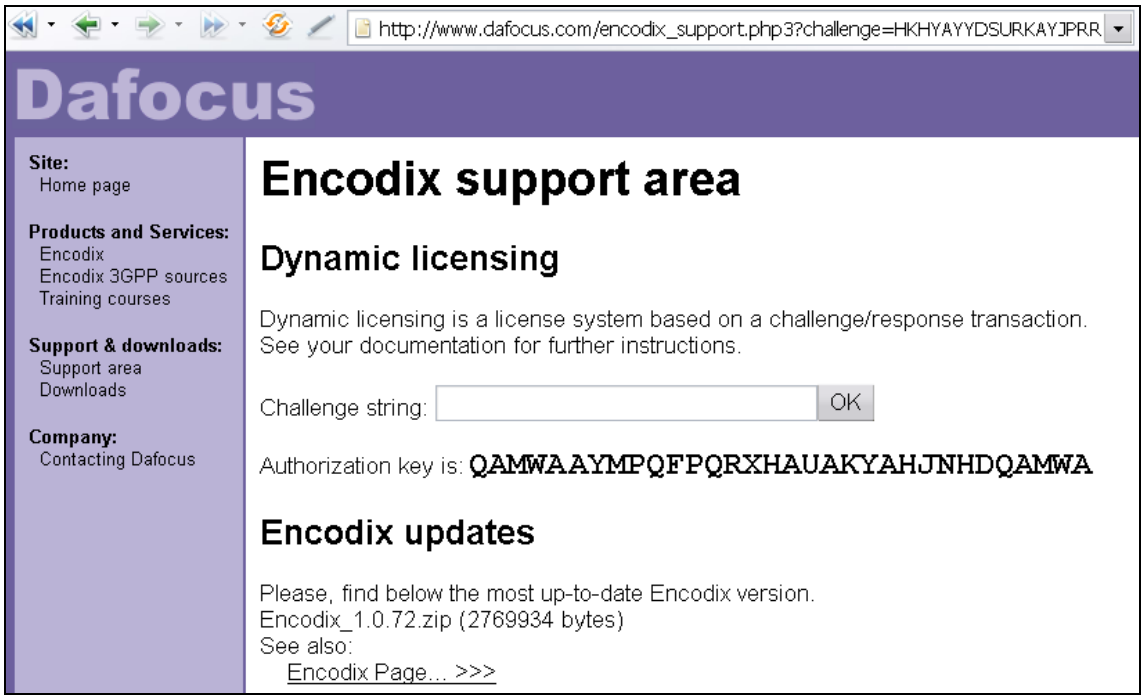
```
http://www.dafocus.com/support/encodix.html
```

This can be achieved by pressing "Connect" on the previous dialog box or by going directly with a internet browser.

The following web page will appear:



Copy the *Challenge string* from the Dafocus license server dialog into the *Challenge string* field in the web page above.
After pressing "OK", the following page appears:



The authorization key can now copied from the web browser and pasted in the license server dialog box.
The license server will not ask again a key unless:

- it is shut down and restarted;
- the PC internal clock is changed;
- the time locked license is expired.

13 Changes

This chapter summarizes changes, bug fixes and enhancements to Encodix TLV modules.

Version 1.0.01 – 20 December 2000
- Added support to conditional and multiple “remainder” fields in BitFields.
- Sequences can now be made of simple types (like integers, octets, etc.).
- Added “size” specification in sequences.
- Added “son of” specification.

Version 1.0.02 – 27 December 2000
- Added support for source-level parameterised message sets

Version 1.0.03 – 3 January 2001
- Added support for BF_Super_Field (see chapter on bit-fields, example 6)
- Shift command (see chapter on bit-fields, example 7)

Version 1.0.04 – 6 January 2001
- Fixed bug (missing “`#define DATA`” declaration in TV and TLV decoding function).

Version 1.0.05 – 6 January 2001
- Added support for inclusion in multiple message set
- Added support for binary fields in sub-fields

Version 1.0.06 – 10 January 2001
- Fixed bug in Pseudo L2 Length encoding: length was encoded in bits 1-8 instead of 3-8.
- Added `PLenAutoFill` keyword.
- Incoming-only and outgoing-only message sets.

Version 1.0.07 – 23 January 2001
- Added preliminary support to CSN.1
- Now keyword “`--EndOfL2Length--`“ has been modified into “`**EndOfL2Length**`“ to avoid conflicts with CSN.1 double-dash comment symbol (`--`).

Version 1.0.08 – 24 January 2001
- Updated generation of `DecodeAndSend_xxx` functions: now they check return value of decoding functions and return `ED_FALSE` in case of error.

Version 1.0.09 – 28 January 2001
- Completed support for CSN.1 decoding.
- Improved speed.

Version 1.0.10 – 15 February 2001
- Full support of CSN.1 encoding and decoding
- Support for custom-sized CSN.1 fields.

Version 1.0.11 - 8 March 2001
- Increased generation performances

Version 1.0.12 - 27 March 2001
- Fixed a bug in CSN.1: used spare byte was 2F instead of 2B
- Fixed a bug in code generation: sometimes decoding or encoding functions were not generate even if needed
- Enhanced error detection: now errors are detected and propagated at any encoding and decoding stage

Version 1.0.13 - 28 March 2001
- Added multi-file feature

Version 1.0.14 - 6 April 2001
- Added specific support for 04.60

Version 1.0.17 - 15 May 2001
- Added support for “External” data types

Version 1.0.18 - 27 May 2001
- Added some features in CSN.1

Version 1.0.20 - 31 May 2001
- Added “slave” keyword in CSN.1

Version 1.0.21 - 18 Jul 2001
- Fixed some minor CSN.1 bugs

Version 1.0.22 - 10 August 2001
- Added support for Telelogic SDL Suite version >= 4.2

Version 1.0.23 - 14 August 2001
- Added native support for 0/1 ext fields.
- Encodix documentation has been enhanced.

Version 1.0.24 - 01 January 2002
- Added support for ED_C_TIGHT

Version 1.0.25 - 24 January 2002
- Added support for CSN.1 "//", "=" and "!"
- Removed comment "//" (conflicts with CSN.1 "//")
- Fixed references in CSN.1: now they are case-insensitive.
- Added support for "SAVE ... AS" in CSN.1
- Added support for direct "%{...}%" in CSN.1 instead of "ENCODECO_EXPR"

Version 1.0.26 - 17 February 2002
- Added `ED_CHECK_STRUCT_DUPES` option
- Now duplicated fields inside structures are detected

Version 1.0.27 - 28 February 2002
- Fixed error occurring when using `octet_array` without SDL support
- Fixed error when tracing CSN.1

Version 1.0.28 - 29 March 2002
- Added "incoming"/"outgoing" specificators for CSN.1 messages.
- Optimized CSN.1 enco/decoding, removing unnecessary steps.
- Added CSN.1 "val(x)" function.
- Added `USER_C_HEADER` custom definition.
- Enhanced CSN.1 stack declaration: now it can be either global or local.

Version 1.0.29 - 11 April 2002
- Fixed a bug in the CSN.1 grammar that made parsing very slow.

Version 1.0.30 - 12 April 2002
- Fixed a bug in the CSN.1 code generator that made compilation impossible when nested definitions with "val" where used.

Version 1.0.31b - 15 May 2002
- Changed to CODEGENIX version 1.1: now executes PHASEMAGIC instead of AUTOMAGIC.
- Added IEEE 802.11 support.
- Fixed a bug in the CSN.1 code generator: order of encoding attempts for choices was not always consistent.

Version 1.0.32 - 6 August 2002
- Added support for new “DUMP” module

Version 1.0.33 - 23 August 2002
- CSN.1 .c file can be now split in chunks
- Fixed a bug in generation of INIT_xxx for group unions.

Version 1.0.34 - 23 October 2002
- Added extensive error checking for messages too short, unexpected IE, etc.
- Now condition check is done at the end
- Set function now work with length in bits

Version 1.0.35 - 18 November 2002
- Fixed the fact that when a bit-field with a single void item is described, Encodix generates an empty structure.
- Added fields ProtocolDiscriminator and MessageType in set structures (ed_c_recog.h)
- Added the ED_DISABLE_MESSAGE_SIGNATURE configuration flag.
- Fixed the c set recognizer function: it didn't set Type to "xxx_Unrecognized" when the message wasn't recognized

Version 1.0.36 - 28 November 2002
- Added custom validators to data fields during decoding.

Version 1.0.37 - 29 November 2002
- Enhanced the ability to skip unknown IEs on optional fields.

Version 1.0.38 - 01 December 2002
- Enhanced the behaviour of optional fields, marked as not present if erroneous
- Changed some error handling macros

Version 1.0.39 - 17 December 2002
- Fixed macro `ED_HANDLE_CONDITIONAL_IE_MSG_TOO_SHORT`, that was returning `ID_MANDATORY_IE_SYNTAX_ERROR` instead of `ED_CONDITIONAL_IE_SYNTAX_ERROR`.
- Now Encodix generates KnownL3IEs writing IEIs in hex for easier reading.
- Now Encodix writes the message *"Warning, IE with IEI=xxx defined in several different ways..."* displaying the IEI in hex.

Version 1.0. 40 - 05 Feb 2003
- Optimized CSN.1 code generator
- Fixed a bug in looping definitions

Version 1.0.41 - 13 Feb 2003
- Added support for Visual Basic

Version 1.0.42 - 20 Feb 2003
- Fixed a bug in CSN.1 optimizer

Version 1.0.43 - 22 May 2003
- Removed most of the C warnings when compiling generated code

Version 1.0.44 - 11 June 2003
- Removed all the remaining C warnings when compiling generated code
- Added the ED_GENERATE_EXPORT_CALL configuration variable
- Fixed "Match_" functions that didn't work with XTID

Version 1.0.45 - 09 July 2003
- Fixed a bug occurring sometimes when using DUMP module with binary data
- Added support for Telelogic TAU C-Micro environment

Version 1.0.46 - 14 Jul 2003
- Added support for old CSN.1 encoder/decoder generation
- Added the "type-only" modifier to CSN.1

Version 1.0.47 - 29 Jul 2003
- Added support for CSN.1 [<max>] specificators

Version 1.0.48 - 18 Aug 2003
- Fixed a bug which affected decoding of CSN.1 sequences like `<bit>**`

Version 1.0.49 - 22 Aug 2003
- Made CSN.1 syntax error detection much faster;
- Solved an internal parser loop when some syntax errors were detected in the source.

Version 1.0.50 - 27 Aug 2003
- Linux version released
- Now min/max size of "L" information elements is checked
- All the preprocessor directives are now left-aligned

Version 1.0.51 - 02 Sep 2003
- Fixed a bug in the CSN.1 optimizer

Version 1.0.52 - 17 Sep 2003
- Optimized EDCopyBits
- Fixed a bug which prevented recognition of xxNNNNNN messages when "xx" was not 0.

Version 1.0.53 - 03 Oct 2003
- Enhanced recognition: now uses again cases instead of "if" statements
- Removed "OPTIONAL" in non-first 0/1ext items

Version 1.0.54 - 04 Oct 2003
- ED_CSN1_H/ED_CSN1_C environment variables now used for CSN1 enhanced too
- ED_KNOWNIE_H environment variable was not always considered
- Added the ED_KNOWNIE_DB environment variable for the common DB of known IE (for MASTER/GROUP generation)

Version 1.0.55 - 07 Oct 2003
- Fixed a bug in the MASTER/GROUP multifile generation system

Version 1.0.56 - 09 Oct 2003
- Extended the P2 Pseudo Length logic: now it can be set manually according to the ETSI specifications.

Version 1.0.57 - 13 Oct 2003
- Released module "Access".

Version 1.0.58 - 15 Oct 2003
- Now CSN.1 generate an "octet" where bitsize is 8 or less; this can be disabled with setting `ED_CSN1_INTEGERS` to 1.

Version 1.0.59 - 16 Oct 2003
- Fixed a bug in the CSN.1 code generator

Version 1.0.60 - 20 Oct 2003
- Added the `ED_01EXT_MULTI_OPTIONAL` configuration option.
- Added the "repeat" keyword to 0/1ext fields.
- Now Access Module generates a list containing all the registered TEDClassBase objects: this allows scanning of existing types.
- Access Module supports the new "TEDClassEnumerator" data type, activated with the same "enum" syntax seen in Dump module.
- Extension .c/.h for C and C++ source files are now programmable.

Version 1.0.61 - 12 Nov 2003
- Length in TLV fields are now allowed up to 9999 octets instead of 999.
- Message type and protocol discriminator are now checked for incoming message sets. They are checked also against their mask if declared with some "xx" set.
- ED_HANDLE_MANDATORY_IE_SYNTAX_ERROR and similar macros are now terminated by ";"
- Added the ED_SHORT data type: now 9-16 bits integers are generated of that type.
- The "SubFields.c/.h" file name is now programmable through macros ED_SUBFIELDS_C/ED_SUBFIELDS_H.
- Added the new Dynamic Data Module: see documentation above for details.
- Now c structures are sorted to enhance efficiency.
- ED_BOOLEAN has been changed to "unsigned char"
- The ED_C_TIGHT is now true by default.
- TI_value is generated as unsigned char instead of integer.
- Some redundant memsets have been removed.
- memset calls are now made through the "ED_RESET_MEM" macro.
- memcpy calls are now made through the "ED_MEM_COPY" macro.
- The parser doesn't hang anymore on syntax errors.
- All files included by generated code are now programmable.

Version 1.0.62 - 29 Dec 2003
- Fixed a bug that sometimes prevented corrected skipping/detection of 4-bit tagged optional information elements.

Version 1.0.63 - 02 Jan 2004
- Fixed a bug in CSN.1 code generator. Situations like `<A>::=<y:B>**; <B>::=<x:1>|0;` ended up with an error.

Version 1.0.64 - 13 Jan 2004
- Fixed a bug in octet range (n-m) in bitfields.

Version 1.0.65 - 08 Mar 2004
- Added the zbit-field declaration, where bits are counted from 0 instead of 1.
- Enco/deco parameters
- Source/Destin are defined as THIS in encode/decode functions (can be disabled with "set ED_GENERATE_THIS 0")
- New "TLVSet" data type.
- Now ED_IS_DECODING or ED_IS_ENCODING are defined together with "DATA" inside conditional expressions
- Support for 24.011 SM-RL messages
- Support for 23.040 messages

Version 1.0.66 - 10 Mar 2004
- Added ED_LIBH_ED_DYNAMIC, ED_LIBH_ED_ACCESS, ED_LIBH_ED_DUMP and ED_LIBH_ED_DUMP_VAL to allow configuration of ed_dynamic.h, ed_access.h, ed_dump.h and ed_dump_val.h.

Version 1.0.67 - 10 Mar 2004
- Fixed some bugs in 23.040 encoding/decoding.

Version 1.0.68 - 11 Mar 2004
- Fixed a bug in XTID generation
- Added configuration keyword ED_KNOWNL3IES to change the name of the variable KnownL3IEs

Version 1.0.69 - 05 Apr 2004
- Fixed some missing "const" in encode functions

Version 1.0.70 - 28 Apr 2004
- Added support for Telelogic TAU SDL Suite 4.4 and 4.5, CAdvanced and CMicro.
- Added Encodix version printout in generated .c/.h files; can be disabled with "set ED_DISABLE_ENCODIX_VERSION 1"

Version 1.0.71 - 11 Jun 2004
- Added the CSN1_ALLOW_FIXED_SEQUENCES setting, that allows CSN.1 to generate fixed sized structures when possible.

Version 1.0.72 - 09 Oct 2004
- Added support for identifiers and C inline code in SAVE AS CSN.1 statement
- Fixed a code generation error when using CSN.1 tags
- Fixed a bug when using in "`val(x)`" (CSN.1) a 'x' label which appears more than once.
- Fixed a bug which prevented compilation of generated code for recognizers using dynamic data.
- Changed default for unspecified conditionals: now they are expected missing.

Version 1.0.73 - 21 Oct 2004
- Added `manual-decode`, `manual-encode` and `manual-encodeco` keywords for ASN.1 definitions.

Version 1.0.74 - 11 Nov 2004
- Fixed a bug when setting default values of void data in 0/1ext fields.

Version 1.0.75 - 21 Mar 2005
- Changed behaviour: conditional tagged IEI are now generated as "optional" if no condition is specified. This feature can be disabled by setting `ED_CONVERT_UNSPECIFIED_CONDITIONS_TO_OPTIONAL` to 0.

Version 1.0.76 - 27 Mar 2005
- Fixed a bug: CSN.1 strings like
  `<foo> ::= 0 | 1 <myValueA: bit(36)> <foo>;`
  produced a C source that did not compile.

Version 1.0.77 - 31 Jul 2005
- Changed the CSN.1 code generator; now it is table based and produces much smaller code.
- Support to the linux version is temporarily suspended.

Version 1.0.78 - 01 Aug 2005
- Changed an heuristic rule of CSN.1: now the label count for the "NULL" entry is -1. Therefore, the NULL statement is always the last one to be parsed.

Version 1.0.79 - 28 Aug 2005
- Fixed a bug that blocked execution when using certain file paths in groups.
- Fixed an error in generating .dep file when file names where specified with backslashes instead of slashes.
- Reactivated the linux version.

Version 1.0.80 – 10 Oct 2005
- Added the ED_CSN1_PREFIX variable to allow linking multiple CSN.1 files.

Version 1.0.81 – 31 Oct 2005
- Added a feature that adds an underscore (_) in front of every CSN.1 name that starts with a number. This can be disabled by setting `ED_CSN1_DISABLE_UNDERSCORE` to 1.
- Setting `ED_GENERATE_SET_ENCODE` to 1 now generates the "SetEncode" functions
- "`GROUPNAME`" is substituted in `ED_CSN1_PREFIX` too
- Fixed some warnings in generated code

Version 1.0.82 – 11 Nov 2005
- Added a feature that adds an underscore (_) in front of every CSN.1 name that starts with a number.

Version 1.0.83 – 3 Dec 2005
- Enhanced the CSN.1 heuristic rules to handle situations where multiple sources for "`val(x)`" functions were found.

Version 1.0.84 – 12 Apr 2006
- Fixed a bug in generation of dynamic ALLOC data macros when DYNAMIC was disabled. Number of bits was set to zero always.

Version 1.0.85 – 17 May 2006
- Fixed: bit-fields declared of fixed size N but specifying only a subset of the bits resulted shorter at the checks.

Version 1.0.86 – 30 Oct 2006
- Added support for CSN.1 "optval" function
- Added the block-wide CSN.1 "slave" keyword
- Added the "s := x .. y" CSN.1 constraints
- Added the ED_CSN1_DISABLE_INTERSECTION TCL variable
- Added the "-" set exclusion symbol (even for single (-011) or multiple (-{000|001|010}) sets.
- Added the parametric CSN.1 definitions
- Enhanced the support for complex CSN.1 expressions
- Added "tcp-lind" "Length Indicator" messages
- Added "udp" messages

Version 1.0.87 - 20 Nov 2006
- Changed ED_CSN1_DISABLE_INTERSECTION into ED_CSN1_DISABLE_SUBCLASSING
- New "compact" code generator. Code is much, much smaller!
- Improved CSN.1 generation speed.
- Now most of CSN.1 2.0 standard is supported
- CSN.1 code generator now supports “callback” mode for data feeding.

Version 1.0.88 - 28 Nov 2006
- Added ED_CSN1_LABEL

- Added support for the (m..n) ranges in CSN.1
- Added support for the (n)# and the (n)// CSN.1 symbols
- Added CSN.1 reference overloading (same definitions with different parameters counts)
- Fixed the division symbol (/) which wasn't supported in the expressions
- Now every CSN.1 number can be expressed, besides of decimal form, in hex (0hNNN or 0xNNN) or binary (bNNNN or 0bNNNN).
- CSN.1 subranges now have a native instruction supporting them.
- Now the CSN.1 custom expressions can return: 0=never loop CSN1C_EXPR_INFINITE=infinite CSN1C_EXPR_BACKTRACK=execute a backtrack CSN1C_EXPR_FAIL=terminate with failure

Version 1.0.89 - 30 Nov 2006

Version 1.0.90 - 05 Dec 2006
- Added CSN.1 support for labels in multiple branches of intersections

Version 1.0.91 - 07 Dec 2006
- Now CSN.1 <octet := 0xNN> is considered as <xxxxxxxx>

Version 1.0.92 - 20 Dec 2006
- Optimized the CSN.1 code generator with the CSN1C_LABEL_VALUE command
- Now the CSN.1 callback function can handle optional fixed values
- Enhanced detection of CSN.1 errors by "!" symbol

Version 1.0.93 - 21 Dec 2006
- Fixed a bug in csn1clib.c

Version 1.0.94 - 22 Dec 2006
- Changed the behaviour of csn1clib.c when non-terminal labels have data (father feeding)

Version 1.0.95 - 27 Dec 2006
- Fixed a bug in the CSN.1 code generator, when repeating references where generated.

Version 1.0.96 - 29 Dec 2006
- Fixed a bug in the CSN.1 code generator, when decoding infinite loops

Version 1.0.97 - 07 Jan 2007

Version 1.0.98 - 18 Jan 2007
- Fixed a bug in the CSN.1 code generator, which generated unaligned DefaultProgram_Labels entries.

Version 1.0.99 - 22 Jan 2007
- Added the fnull CSN.1 keyword that matches only if the string is terminated (when decoding).
- Added the ED_CSN1_FORCE_NULLS option

Version 1.0.100 - 23 Jan 2007
- Added the FirstErrorLabelId
- Added the "truncable" bit-field command

Version 1.0.101 - 03 Feb 2007
- Fixed name generation of CSN.1 structures when working with groups

Version 1.0.102 - 08 Feb 2007
- Added the AutoFill command

Version 1.0.103 - 13 Feb 2007
- Added the Continue parameter to TCSN1CContext

Version 1.0.104 - 15 Feb 2007
- Changed the behaviour of tags, that now retain the same persistence of the older CSN.1 code generator

Version 1.0.105 - 17 Feb 2007
- Fixed the wrong generation of ED data in some looping CSN.1 definitions (CSN.1 2.0)

Version 1.0.106 - 19 Feb 2007
- The generation of CSN.1 types with "type-only" was different.

Version 1.0.107 - 18 Jun 2007
- Fixed some warnings in Access classes

Version 1.0.108 - 27 Jun 2007
- Added Get/SetDefaultLabel for TEDClassEnumerated: now it handles the "default" definition of enums.
- Changed the debug print of TEDDataEnumerated: now it prints the enumerations

Version 1.0.110 - 11 Dec 2007
- Fixed some demo programs

Version 1.0.111 - 21 Feb 2008
- Removed CURRMS, that was returning numbers too big

Version 1.0.113 - 29 Feb 2008
- fixed csn1lib.c/.h files to compile with recent GCC compilers
- added demos for 44.060 and 44.018
- added BinaryDump function in bitencode.h/.c

Version 1.0.114 - 23 Jun 2008
- added support for IEEE 802.16
- enhanced dynamic data generation

Version 1.0.115 - 25 Jul 2008
- Further enhanced dynamic data generation

Version 1.0.116 - 12 Aug 2008
- Fixed CSN.1, it was executing ENCODE actions even during decode and vice versa.

Version 1.0.117 - 23 Sep 2008
- Added TCL configuration entries ED_C_ACCESS_POSTFIX and ED_C_TYPE_PREFIX

Version 1.0.118 - 10 Nov 2008
- Added checks on sequence size when decoding CSN.1

Version 1.0.119 - 20 Nov 2008
- Enhanced the demos

Version 1.0.120 - 04 Feb 2009
- Added support for TS24.301

Version 1.0.121 - 09 Mar 2009
- Added support for Linux x86_64
- Linux executables linked statically (no more -ltcl)

Version 1.0.122 - 22 Apr 2009
- Fixed small type incompatibilies

Version 1.0.123 - 27 Apr 2009
- Added 16-bit tags

Version 1.0.124 - 30 Jul 2009
- Fixed a timing value that was interpreted as an invalid octal

Version 1.0.125 - 20 Sep 2010
- Added the 'custom-len-decl' and 'custom-len' keywords.

Version 1.0.126
- Fixed a small bug in 'EDSkipKnownIE'

Version 1.0.127 - 27 May 2011
- Corrected a typo in 24.301: "trasaction" instead of "transaction"

Version 1.0.128 - 13 Aug 2012
- added locator support
- changed CSN1_AUTO_TAG_BASE, now a “localconfig.tcl” setting instead of a C macro
- added "%TAG(n)" in CSN.1 tags
- added "ENCODECO_OPTIONAL"
- added C command "POP_TAGS" for CSN.1

- added C command "EXISTS_TAG" for CSN.1
- added C command "TAGZ"
- added CSN.1 command "LIMIT_SIZE(tag): %{expr}%" and "RESTORE_SIZE(tag)"

Version 1.0.129 - 28 Aug 2012
- added the ability to disable fields sorting

Version 1.0.130 - 02 Sep 2012
- enhanced support for IEEE 802.16
- fixed a possible problem on CSN.1 binary data when decoding badly formed strings

Version 1.0.131 - 27 Dec 2012
- added support for "ID2" messages (i.e. messages with 2-bit message type)