Technical Document

Generic Tool Chain

GTC USERGUIDE

Document Number:	Document number to be assigned
Version:	0.6
Status:	Draft
Approval Authority:	
Creation Date:	2003-Dec-05
Last changed:	2015-Mar-08 by Texas Instruments
File Name:	gtc_userguide.doc

Important Notice

Change History

Date	Changed by	Approved by	Version	Status	Notes
2003-Dec-05	Kerstin Thiemann		0.1	Draft	1
2003-Dec-05	Kerstin Thiemann			Draft	2
2004-Jan-06	Kerstin Thiemann			Draft	3
2004-Jan-23	Kerstin Thiemann			Draft	4
2004-Sep-27	Kerstin Thiemann			Draft	5
2004-Dec-14	Kerstin Thiemann			Draft	6

Notes:

1. Initial version
2. Revision of subchapter Feature Flags
3. Providing tables to facilitate top down access
4. Expanding description of coding types for standard information elements
5. Expanding description of coding types for non-standard information elements
6. Added description of coding types CSN1_CHOICE1 and CSN1_CHOICE2

Table of Contents

List of References

[1.]	**ccd_userguide.doc**	CCD Users´ Guide - TI Internal Technical Document
[2.]	**8343_308_LLD_XML_Representation-004.doc**	SAP/MSG Editor, XML Representation – Low Level Design
[3.]	**8350_300_MSG_Syntax.doc**	TI User Guide - Syntax description for air interface message documents
[4.]	**8350_301_SAP_Syntax.doc**	TI User Guide - Specifying Service Access Points
[5.]	**6368_807.doc**	TI Memo Feature Flag Catalogue
[6.]	**sij_memo_ff_usg.doc**	TI Memo: Feature Flag and Their Support by GEN Tool Chain
[7.]	**Requirement_and_Specification_FeatureFlags_fr2sbk1.doc**	GPF Tools: Feature Flags - Requirements and Specification
[8.]	**ETSI TS 100 550 (GSM 04.01)**	Mobile Station - Base Station System (MS - BSS) interface; General aspects and principles
[9.]	**3GPP TR 21.905**	Vocabulary for 3GPP Specifications
[10.]	**3GPP TR 23.101**	General UMTS Architecture
[11.]	**3GPP TS 24.007**	Mobile radio interface signalling layer 3; General Aspects
[12.]	**3GPP TS 24.008**	Mobile radio interface layer 3 specification Core Network Protocols-Stage 3
[13.]	**3GPP TS 24.011**	Point-to-Point (PP) Short Message Service (SMS) support on mobile radio interface
[14.]	**3GPP TS 44.018**	Mobile radio interface layer 3 specification; Radio Resource Control Protocol
[15.]	**3GPP TS 44.060**	General Packet Radio Service (GPRS); Mobile Station (MS) - Base Station System (BSS) interface; Radio Link Control/Medium Access Control (RLC/MAC) protocol
[16.]	**CCITT Recommendation X.200**	Reference Model of Open Systems Interconnection for CCITT Applications"
[17.]	**CCITT Recommendation X.210**	Open Systems Interconnection layer service definition conventions"
[18.]		XML Schema Part 0: Primer, W3C Recommendation, 2 May 2001
[19.]		XML Schema Part 1: Structures, W3C Recommendation, 2 May 2001
[20.]		XML Schema Part 2: Datatypes, W3C Recommendation, 2 May 2001

Abbreviations

MDF	Message Description File
MSG	Message
PDF	Primitive Description File
IE	Information Element
IEI	Information Element Identifier
SAP	Service Access Point

Other abbreviations used in the present document are listed in 3GPP TR 21.905 [9.].

0 Scope

This document shall assist developers to write AIR Message and Service Access Point description documents which can be processed by the TI tool chain. These description documents and the intermediate results within the tool chain itself provide a basis for various other TI tools (test tools, tracing tools, programming tools etc.).

The aim of this document is to explain how to define AIR messages and service access points (SAPs). This document will describe how AIR messages and service access points' descriptions must be structured and how the different elements can be combined. The TI tool chain demands these descriptions to be of a specific format.

Finally this document will describe the mapping from the AIM/SAP into the resulting C code and describe some of the features in more detail. This will also include an overview of combinations of elements, columns etc., which are commonly used. This document does in some cases illustrate some points by use of C examples. Such examples will, however, be subject to changes in case of alterations to the TI coding standard.

1 Introduction

A GSM/GPRS PLMN supports a wide range of services, which a user accesses by a standard set of interfaces at a mobile station (MS). A basic architectural split is between the user equipment (terminals) and the infrastructure. This results in two domains: the **User Equipment Domain** and the **Infrastructure domain** (PLMN).

User equipment is the equipment used by the user to access GSM/GPRS services. User equipment has a radio interface to the infrastructure. The infrastructure consists of the physical nodes, which perform the various functions required to terminate the radio interface and to support the telecommunication services requirements of the users. The infrastructure is a shared resource that provides services to all authorized end users within its coverage area.

The user equipment is also referred to as mobile station consisting of the physical equipment used by a PLMN subscriber. It is an entity composed of the **Mobile Equipment Domain** (ME) and the **GSM Subscriber Identity Module** (SIM). User Equipment is a device allowing a user access to network services. For the purpose of GSM/GPRS specifications the interface between the UE and the network is the radio interface. The **Infrastructure domain** comprises roughly the functions specific to the access technique and the functions may potentially be used with information flows using any access technique.

The PLMN infrastructure is logically divided into a Core Network (CN) and an Access Network (AN) infrastructures. The CN itself is constituted of a Circuit Switched (CS) domain and a Packet Switched (PS). These two domains are overlapping, i.e. they contain some common entities. A PLMN can implement only one domain or both domains. The CS domain refers to the set of all the CN entities offering "CS type of connection" for user traffic as well as all the entities supporting the related signalling. A "CS type of connection" is a connection for which dedicated network resources are allocated at the connection establishment and released at the connection release. The PS domain refers to the set of all the CN entities offering "PS type of connection" for user traffic as well as all the entities supporting the related signalling. A "PS type of connection" transports the user information using autonomous concatenation of bits called packets: Each packet can be routed independently from the previous one.

1.1 Protocol Modeling Principles

The protocols used to exchange information between the **User Equipment Domain** and the **Infrastructure domain** are specified using the concepts of the reference model of Open System Interconnection (OSI) given in CCITT Recommendations X.200 and X.210.

The basic structuring technique in the OSI reference model is layering. According to this technique, communication among application processes is viewed as being logically partitioned into an ordered set of layers represented in a vertical sequence as shown in Figure 1.

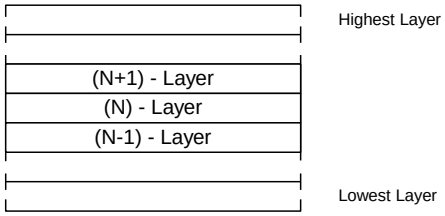


Figure 1: Layering

Entities exist in each layer. Entities in the same layer, but in different systems, which must exchange information to achieve a common objective, are called "peer entities". Unidirectional radio link for the transmission of signals from an **Infrastructure domain** access point to a UE (also in general the direction from Network to UE) is called **Downlink**. An **Uplink** indicates the contrary information flow: It is a unidirectional radio link for the transmission of signals from a **User Equipment Domain** to an **Infrastructure domain** access point (also in general the direction from UE to Network).
Entities in adjacent layers interact through their common boundary. The services provided by the (N + 1) - layer are the combination of the services and functions provided by the (N) - layer and all layers below the (N) - layer[1]. Layer-to-layer interactions are specified in terms of service primitives. The primitives represent, in an abstract way, the logical exchange of information and control between adjacent layers. They do not specify or constrain implementation. Primitives are also used to describe information exchange between layers and the mobile management entity.

The primitives that are exchanged between the (N + 1) - layer and the (N) - layer are of the following four types (see Figure 2).

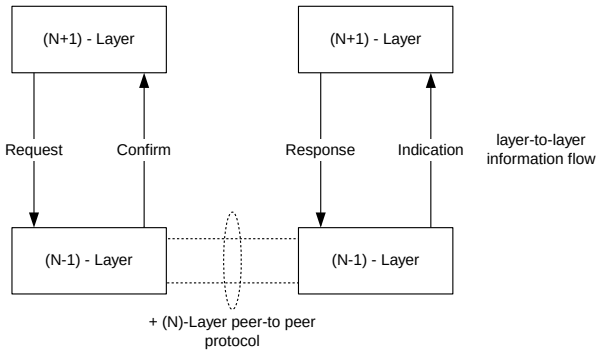


Figure 2: Primitive Action Sequence for Peer-to-Peer Communication

The REQUEST primitive type is used if a higher layer is requesting a service from the next lower layer. A layer uses the INDICATION primitive type providing a service to notify the next higher layer of activities related to the primitive type REQUEST. To acknowledge a primitive type INDICATION sent from a lower to an upper layer the receiving layer uses the RESPONSE primitive type. The CONFIRM primitive type is used by the layer providing the requested service to confirm that the activity has been completed.

1.2 Peer-to-Peer Messages

In the world of GSM/GPRS/UMTS the peer-to-peer messages exchanged between the User Equipment Domain and the Infrastructure domain are transmitted over the air-interface. For GSM/GPRS/UMTS

[1] Management functions may also be required. They may include functions which are common for several layers and are not supported by the services provided by a specific layer. Examples of such functions are error reporting, status reporting and management of the operation of certain layers. Such management functions do not require that peer-to-peer messages are sent across the network interfaces.

protocols, these messages are bit strings of variable length, formally a succession of a finite, possibly null, number of bits (i.e., elements of the set {"0", "1"}), with a beginning and an end. These messages are standardized by ETSI/3GPP, and thus have well defined formats.

Data in GSM protocol stack entities are normally hold in C-structures. Currently TI Air Interface Messages are documents written in XML as part of the high-level design phase. In the past TI Air Interface Messages are defined through a Microsoft Word document. When air interface messages are needed in actual code, the description documents run through the TI tool chain (cf. GTC Generic Tool Chain), which produce header files and other data needed in program code.

The size of messages sent over the air interface is reduced to a minimum so as to enable rapid and compact transfer. Messages are defined as a structure of information elements concatenated as a bit stream.

Microprocessors are capable of rapid memory access, which is not bit-orientated, but rather byte-, word- or long -orientated. In addition, some processor families allow access to even addresses only.

In general, air-interface messages do not start at byte boarders and are not multiples of eight bits in length. Incoming message must be decoded, in other words, quickly transformed into a format that the target system can read (e.g. C-structure). Outgoing messages must be encoded from a C-structure to a bit stream.

1.3 Layer-to-Layer Communication

In the present document, the communication between adjacent layers and the services provided by the layers are distributed by use of abstract service primitives. Primitives consist of commands and their respective responses associated with the services requested of another layer.

This document contains a description of the way service access points are specified for entities in the TI protocol stacks. The definition of a service access point is based on the layered protocol stack model. A service access point identifies the services provided by an entity to other entities placed at a higher level in the protocol stack, either in a higher layer or with hierarchically higher rank within the same layer.

A service access point completely defines the interface to be used to gain access to a set of services provided by an entity. The definition of the interface can be based on a set of primitives and as extension to OSI a set of function calls or both, depending on the nature of the services provided by the entity.

In TI terms, a service access point is defined through a XML document, newly. Like TI AIR Messages service access point definitions are written as Microsoft Word document up to now. These documents contain descriptions of the service access points in a specific format, which can be processed by the TI tool chain, resulting in output for the various TI tools (test tools, tracing tools, programming tools etc.).

1.4 GTC Generic Tool Chain

The Generic TI Tool Chain (GTC) is a subset of the tool chain belonging to the protocol stack development process (see Figure 3). There, a blue line surrounds the affected part.

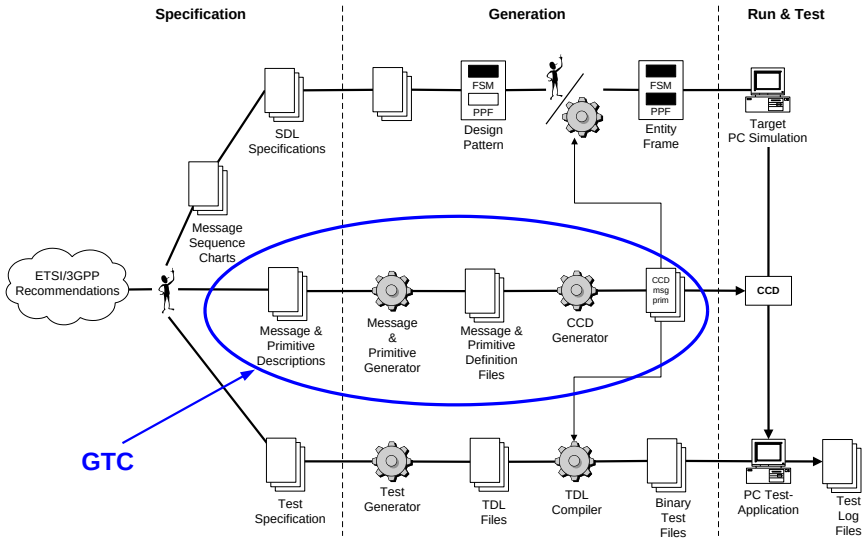


Figure 3: Protocol Stack Development Methods

The ETSI / 3GPP specifications are the initial point of the protocol stack development process. These specifications standardize the peer-to-peer messages between a MS and its network peer, which have well defined formats. Inter layer communication is not subject of any similar strict regulation. The interaction between two adjacent layers are described in terms of primitives where the primitives represent the logical exchange of information and control between these layers. The primitives do not specify or constrain implementations. The specifications provide a statement of requirements concerning generic names and parameters; services provided to upper layers and services expected from lower layers are specified as well.

The developer is assigned to write AIR Message and Service Access Point description documents which can be processed by the TI tool chain, resulting in output for the various TI tools (test tools, tracing tools, programming tools etc.). The GTC tool chain supports two description formats. Currently AIM/SAP description documents (*.aim/*.sap) are written in XML as part of the high-level design phase. These XML descriptions replace the old format (*.doc). In the past they are defined through a Microsoft Word document.

SAPE is an environment for creating and editing of AIM and SAP documents based on XML and to convert such documents into the pdf/mdf format. It consists of an Editor for the underlying XML representation and a converter from xml (*.sap, *.aim) to pdf/mdf. *.mdf or *.pdf files are intermediate file formats appropriate for the next tool *ccdgen* with the purpose of message and primitive definitions, respectively. The SAPE editor is based on the Eclipse platform and is implemented as a plug-in for this platform while the converter uses the Xalan Xslt implementation.

In case of descriptions using Microsoft Word document format *xGen100* generates the intermediate *.mdf or *.pdf files (See readme_xGen100.txt for further information about *xGen100* that transforms MSG and SAP documents to the intermediate files).

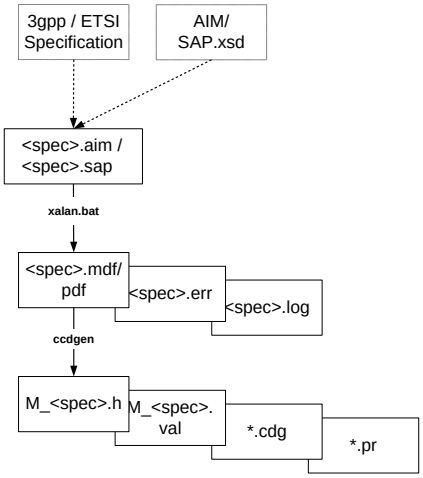


Figure 4: Workflow of the Generic TI Tool Chain

CCDGEN (TI Coder Decoder Generator) is a compiler that transforms message description files (*.mdf) or primitive description files (*.pdf) to tables (*.cdg). The message description tables are the essential input for the TI Coder Decoder CCD to encode structured message to bit-streams and to decode bit-streams to structured message. The primitive description tables are needed by PCON. PCON is primitive converter that has to be called in the test interface drivers to achieve platform independent message format for the test interface communication. In addition CCDGEN generates C header files which contain C structures. These structures describe messages or primitives.

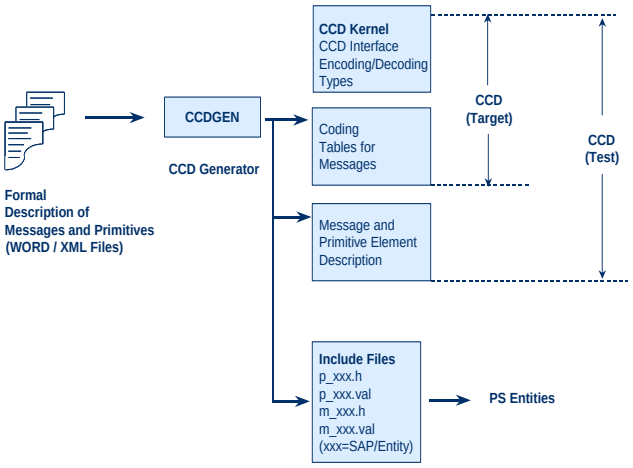


Figure 5: CCDGEN Functionality

To encode bit stream-messages from data in C-structures and vice versa there is the TI Coder Decoder CCD. Additionally there are some functions included to code and decode simple data types like byte and long. The next part gives a more detailed overview of CCD functionality.

1.5 CCD in Overview

CCD is an interpreter which uses an optimised database for high performance. The database contains the rules for coding and decoding all GSM, GPRS or UMTS air-interface messages. It is also possible to use CCD database to represent messages in readable form. Applications using CCD are test systems, tools for analysing signalling and the protocol stack running on a certain target.

A language for describing the messages to generate the CCD database has been defined, which rules over the syntax of MDF and PDF files. The CCD compiler (ccdgen.exe) compiles these descriptions outside the Protocol Stack at generation time. The compiler generates the database for CCD and the structure definitions (C-header files) used by the Protocol-Stack components.

The interface to the applications, i.e. the Protocol-Stack components, is very simple, consisting of an encoding and a decoding function. Thus, all encoding or decoding is carried out by a single function call to CCD. The interface offers also functions to retrieve information on errors occurred while encoding or decoding procedures.

Primitives are used for communication between Protocol-Stack components. They are defined as C-structures, in the same way as messages. The CCD compiler generates the C-structures in the form of header files used by the Protocol Stack components. All primitives are defined with a description language. The CCD Compiler (CCDGEN) generates header files with C-Structures and constants which are included in the source code of the Protocol-Stack entities.

The second types of information carriers are messages according to the GSM, GPRS or UMTS standard. The messages are coded as bit streams and are outside the target system visible at the air-interface. To handle messages efficiently they must be converted from a bit stream to a C-Structure and vice versa. CCD carries this out on the target system. Encoding of a GSM message means the transformation of a C-structure to a bit stream according to the protocol specifications. Decoding means the transformation from a bit stream to a C-structure.

Convert bit-oriented air message to byte-oriented C structure and vice versa

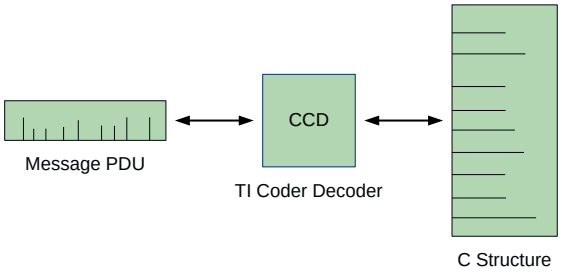


Figure 6: CCD Functionality

2 Message and Primitive Editorial Description Catalogues - SAPE

As it must be possible for the TI tool chain to process documents containing AIM / SAP specifications, the structure of the document is standardized. If the structure of the document is incorrectly implemented, the tool chain will not recognize the document as an AIM respectively SAP specification during translation.

The Air Interface Message and the SAP description XML documents are created based on an XML schema by using the SAPE tool and choosing the appropriate file type (either AIM for Air Interface Message or SAP for Service Access Point description) when creating a new document. Writing AIM/SAP specifications by **SAPE does not require any knowledge about XML language and associated members of the XML family in detail. The SAPE graphical user interface cares about the transformation of table data to well-formed and valid XML data.**

As well it is possible to write an XML document by using a text editor, because XML (Extensible Markup Language) is a simple, very flexible text format. But, the TI tool chain demands for special XML constructs. The following section is not indented to explain the XML language and associated members of the XML family in detail. It is assumed that the reader being interested in AIM/SAP XML representations is familiar with Extensible Markup Language (XML), XML schema description language and certain terms that will be used in conjunction with XML.

The XML description traces back to the concepts of well formedness and validity. It is very simple to check a document for well formedness, while validation requires that a document follows the constraints expressed in its XML schema definition. The XML schema definition provides the rough equivalent of a context-free grammar for a document type. The AIM/SAP schemata define and describe a certain class of XML documents by using special constructs to constrain document structure (order, occurrence of elements, attributes). If the user writes an AIM or SAP XML document by a text editor it is mandatory that

- the document has to match the logical and physical structures which must be nested properly (→well-formed) and
- the document conforms to the particular schema *.xsd which determines the appearance of certain elements, their content and the appropriate order in the instance document (→valid).

In cases of using the SAPE editor this tool validates that the XML document conforms to the rules described in the associated schema. The user needs not to care about the order and the form of any structures. The SAPE editor offers a list of possible sub-elements and their associated and allowed actions. In the XML representation there isn't any section numbering present.

Due to the fact that SAP and AIM documents contain a large amount of different elements with different meanings, this section is broken into several subsections. There are common elements in both types of documents and elements that are specific for the purposes of the different types of interfaces that they describe (either SAP or AIM). One section deals with the common elements which could be part of SAP **and** AIM documents, and there is one section each for specific elements which are present in one kind of document only

These sections are further separated to associate a group of SAPE tables belonging to one main topic with the appropriate XML structures. Each of these subsections explains the purpose of each provided table and its columns in conjunction with the XML node element and its child elements. It might be easier to understand this part of the document if you start reading the sections belonging to the root elements SAP and AIM respectively. All sub-elements are linked with each other. This will help to navigate through the element structure.

The following two tables provide an overview of all parts (common and specific) belonging to AIMs (Air Message Specification) respectively to SAPs (Service Primitive Specification). Each table shows the hierarchical order of nested elements. All elements are linked to the appropriate document parts. The common elements, AIM specific elements and SAP specific elements are shaded in different manner to facilitate orientation. These tables are intended to support an easy top down access to the information overload.

AIM - Air Interface Message

- 2.1.2.1 Document Information Section
  - 2.1.3.2 DocName
  - (2.1.4) DocNum
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.1.1.2.1 Document History
    - 2.1.1.1 DocVersion
    - (2.1.4) Date
    - (2.1.4) DocStatus
  - (2.1.4) DocRef
- 2.1.2.2 Pragmas Section
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.1.2.2.1 Pragma
    - 2.1.3.4 Name
    - (2.1.4) Value
  - 2.1.1.2 History
    - (2.1.4) Date
- 2.1.2.3 Constants Section
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.1.2.3.1 Constant
    - 2.1.3.1 Alias
    - 2.1.3.5 ItemLink
      - 2.1.3.2 DocName
      - 2.1.3.4 Name
    - (2.1.4) Value
    - 2.1.3.6 Feature Flags
    - 2.1.3.3 Group
  - 2.1.1.2 History
    - (2.1.4) Date
- 2.2.1 Messages Section
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.2.1.1 Message
    - 2.1.1.1 Description
      - 2.1.1.1.1 Listing element
      - 2.1.1.1.2 Linked document
        - 2.1.3.4 Name
    - 2.2.1.1.1 Message Definitions
      - 2.1.3.4 Name
      - (2.2.5) MsgID
      - (2.2.1.1.1) MsgLenMax
      - 2.1.3.6 Feature Flags
      - 2.1.3.3 Group
    - 2.2.1.1.2 Message Items
      - 2.1.3.5 ItemLink
        - 2.1.3.2 DocName
        - 2.1.3.4 Name
      - 2.1.3.1 Alias
      - 2.2.4.2 Type
        (2.2.5) ItemTag
        2.2.4.1 Control
        - 2.2.4.1.1 Type Modifier Element
        - 2.2.4.1.2 Condition Element
        - 2.2.4.1.3 Command Sequence Element
        - 2.2.4.1.4 BitGroupDefinition Element
      - 2.1.3.6 Feature Flags
    - 2.1.1.2 History
      - (2.1.4) Date
- 2.2.2 Structured Elements Section (see next page)

- 2.2.2 Structured Elements Section
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.2.2.1 Structured Message Elements
    - 2.1.1.1 Description
      - 2.1.1.1.1 Listing element
      - 2.1.1.1.2 Linked document
        - 2.1.3.4 Name
    - 2.2.2.1.1 Structured Element Definitions
      - 2.1.3.4 Name
      - 2.1.3.6 Feature Flags
      - 2.1.3.3 Group
    - 2.2.2.1.2 Structured Element Items
      - 2.1.3.5 ItemLink
        - 2.1.3.2 DocName
        - 2.1.3.4 Name
      - 2.1.3.1 Alias
      - 2.2.4.2 Type
      - (2.2.5) ItemTag
      - 2.1.3.7 UnionTag
        - 2.1.3.4 Name
        - (2.1.4) Value
      - 2.2.4.1 Control
        - 2.2.4.1.1 Type Modifier Element
        - 2.2.4.1.2 Condition Element
        - 2.2.4.1.3 Command Sequence Element
        - 2.2.4.1.4 BitGroupDefinition Element
      - 2.1.3.6 Feature Flags
    - 2.1.1.2 History
      - (2.1.4) Date
- 2.2.3 Basic Elements Section
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.2.3.1 Basic Message Elements
    - 2.1.1.1 Description
      - 2.1.1.1.1 Listing element
      - 2.1.1.1.2 Linked document
        - 2.1.3.4 Name
    - 2.2.3.1.1 Basic Element Definitions
      - 2.1.3.4 Name
      - 2.1.3.6 Feature Flags
      - 2.1.3.3 Group
    - 2.1.3.8 ValuesLink
      - 2.1.3.2 DocName
      - 2.1.3.4 Name
    - 2.1.1.2 History
      - (2.1.4) Date
- 2.1.2.4 Substitutes Section
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.1.2.4.1 Substitute
    - 2.1.3.1 Alias
    - 2.1.3.5 ItemLink
      - 2.1.3.2 DocName
      - 2.1.3.4 Name
    - 2.1.3.6 Feature Flags
  - 2.1.1.2 History
    - (2.1.4) Date
- 2.1.2.5 Values Section (see next page)

2.1.2.5 Values Section				
	2.1.1.1 Description			
		2.1.1.1.1 Listing element		
		2.1.1.1.2 Linked document		
			2.1.3.4 Name	
	2.1.2.5.1Values			
		2.1.1.1 Description		
			2.1.1.1.1 Listing element	
			2.1.1.1.2 Linked document	
				2.1.3.4 Name
		2.1.2.5.1.1 ValuesDef		
			2.1.3.4 Name	
			2.1.3.6 Feature Flags	
			2.1.3.3 Group	
		2.1.2.5.1.2 ValuesItem		
			(2.1.4) Value	
			2.1.3.1 Alias	
			2.1.3.6 Feature Flags	
		2.1.2.5.1.3 ValuesRange		
			2.1.3.1 Alias	
		2.1.2.5.1.4 ValuesDefault		
			(2.1.4) Value	
		2.1.1.2 History		
			(2.1.4) Date	
2.1.2.6 Annotations Section				
	2.1.2.6.1 Annotation Element			
		2.1.2.6.2 Data Target		

Table 1: Air Interface Message Structure

Common Elements

AIM Specific Elements

SAP – Service Primitive Specification

- 2.1.2.1 Document Information Section
  - 2.1.3.2 DocName
  - (2.1.4) DocNum
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.1.1.2.1 Document History
    - 2.1.1.1 DocVersion
    - (2.1.4) Date
    - (2.1.4) DocStatus
  - (2.1.4) DocRef
- 2.1.2.2 Pragmas Section
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.1.2.2.1 Pragma
    - 2.1.3.4 Name
    - (2.1.4) Value
  - 2.1.1.2 History
    - (2.1.4) Date
- 2.1.2.3 Constants Section
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.1.2.3.1 Constant
    - 2.1.3.1 Alias
    - 2.1.3.5 ItemLink
      - 2.1.3.2 DocName
      - 2.1.3.4 Name
    - (2.1.4) Value
    - 2.1.3.6 Feature Flags
    - 2.1.3.3 Group
  - 2.1.1.2 History
    - (2.1.4) Date
- 2.3.1 Primitives Section
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.3.1.1 Primitive
    - 2.1.1.1 Description
      - 2.1.1.1.1 Listing element
      - 2.1.1.1.2 Linked document
        - 2.1.3.4 Name
    - 2.3.1.1.1 Primitive Definitions
      - 2.1.3.4 Name
      - 2.3.5.1 Primitive Identifier
      - 2.1.3.6 Feature Flags
      - 2.1.3.3 Group
    - 2.3.1.1.2 Primitive Items
      - 2.1.3.5 ItemLink
        - 2.1.3.2 DocName
        - 2.1.3.4 Name
      - 2.1.3.1 Alias
      - 2.3.5.2 Control
      - 2.1.3.6 Feature Flags
    - 2.1.1.2 History
      - (2.1.4) Date
- 2.3.2 Structured Elements Section (see next page)

- 2.3.2 Structured Elements Section
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.3.2.1 Structured Primitive Elements
    - 2.1.1.1 Description
      - 2.1.1.1.1 Listing element
      - 2.1.1.1.2 Linked document
        - 2.1.3.4 Name
    - 2.3.2.1.1 Structured Primitive Element Definitions
      - 2.1.3.4 Name
      - 2.1.3.1 Alias
      - 2.1.3.6 Feature Flags
      - 2.1.3.3 Group
    - 2.3.2.1.2 Structured Primitive Element Items
      - 2.1.3.5 ItemLink
        - 2.1.3.2 DocName
        - 2.1.3.4 Name
      - 2.1.3.1 Alias
      - 2.1.3.7 UnionTag
        - 2.1.3.4 Name
        - (2.1.4) Value
      - 2.3.5.2 Control
      - 2.1.3.6 Feature Flags
    - 2.1.1.2 History
      - (2.1.4) Date
- 2.3.3 Basic Elements Section
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.3.3.1 Basic Primitive Elements
    - 2.1.1.1 Description
      - 2.1.1.1.1 Listing element
      - 2.1.1.1.2 Linked document
        - 2.1.3.4 Name
    - 2.3.3.1.1 Basic Element Definitions
      - 2.1.3.4 Name
      - 2.3.5.4 Type
      - 2.1.3.6 Feature Flags
      - 2.1.3.3 Group
    - 2.1.3.8 ValuesLink
      - 2.1.3.2 DocName
      - 2.1.3.4 Name
    - 2.1.1.2 History
      - (2.1.4) Date
- 2.1.2.4 Substitutes Section
  - 2.1.1.1 Description
    - 2.1.1.1.1 Listing element
    - 2.1.1.1.2 Linked document
      - 2.1.3.4 Name
  - 2.1.2.4.1 Substitute
    - 2.1.3.1 Alias
    - 2.1.3.5 ItemLink
      - 2.1.3.2 DocName
      - 2.1.3.4 Name
    - 2.1.3.6 Feature Flags
  - 2.1.1.2 History
    - (2.1.4) Date
- 2.3.4 Functions Section (see next page)

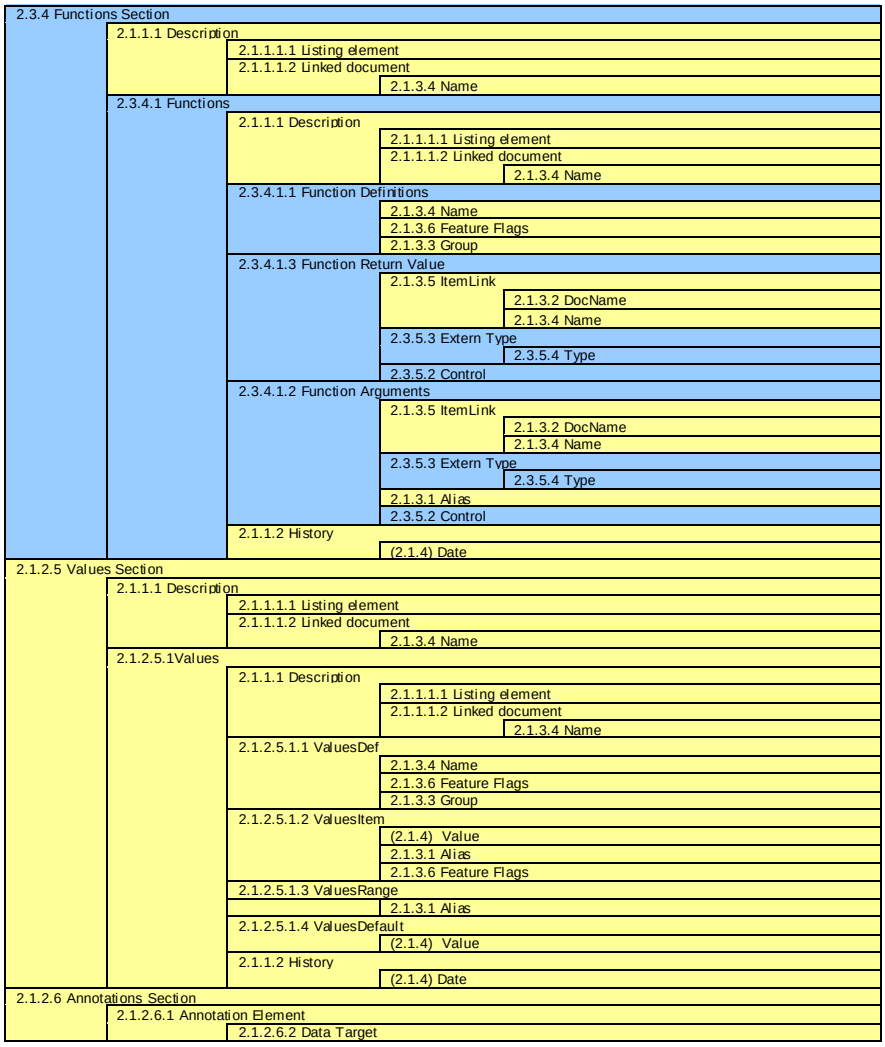


Table 2: Service Primitive Structure

Common Elements

SAP Specific Elements

2.1 Common Description Instruments

This part deals with the common elements which will be used for both types of documents (either SAP or AIM). As a consequence to this the similarities are grouped into the separate, shared schema "sapaim.xsd" that could be used by both document types. The more specific schema files, which are necessary for the different types of interfaces, include this common schema.

The list below shows the common elements belonging to the top level. Some sections may be left out; these sections are marked [optional].

Document Information	Container for all document relevant information
Pragmas [optional]	Used to control and to modify the behaviour of the TI tool chain
Constants [optional]	Contains global constants and used to assign a value to a variable
Substitutes [optional]	Used to define a new name for an existing element
Values [optional]	Used as aliases for user specific values
Annotations [optional]	Container to keep annotations which may belong to any element of the document

Each section above contains a number of subsections, which act as keywords, separating different types of information.

2.1.1 Sharable Subsections

The usage of the items belonging to the sharable subsections is multifaceted. These items appear in conjunction with common, message specific and primitive specific description elements. They are concatenated with different levels in the XML structure. Sharable subsection items may belong to node element as well as to child elements.

2.1.1.1 Description

A Description holds informative data to provide additional facts about the object in the section. It is entirely informational, i.e. the tool chain does not use this part. Therefore, it is not mandatory in any section, but it is strongly recommended that a description is included in any section.

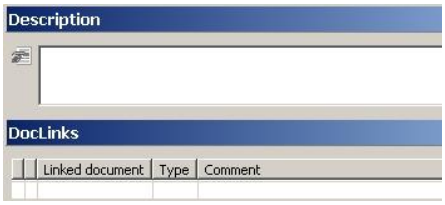


Figure 7: SAPE Description Element

The Description element consists of an indefinite choice of optional child elements:

- The Section element,
  which is provided by the SAPE editor as a field labelled with Description, offers a plain text field to hold any combination of text or digits.
- The Listing element
  is intended for conversion of the old document format (MS Word) to the XML document format. The SAPE editor does not provide this element. For every list that could be identified within descriptive text a *List* element has to be created.

- The DocLink element
  can be found as Linked Description Element. Although SAPE provides a self-contained table the DocLink element belongs to the superordinate Description element. It offers the possibility to link arbitrary files.

This set of child elements have been created as a compromise to the old document format (MS Word) to provide the possibility to format the Description content in a certain way. This is limited to sections, lists and references to other documents, which are mainly intended to be drawings. All of these elements could be mixed in an arbitrary sequence. See the model for the *Description* element:

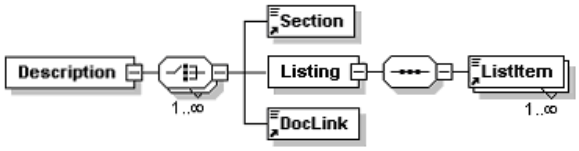


Figure 8: Model of a Description Element

Element Description	
Children	**Section Listing DocLink**
Used by	Elements **ConstantsSection DocInfoSection PragmasSection SubstitutesSection Values Values-Section Message MessagesSection MsgBasicElem MsgBasicElementsSection MsgStructElem MsgStructElementsSection FunctionsSection Function PrimitiveSection Primitive PrimStructElem PrimBasicEle-mentsSection PrimBasicElem**

XML schema	`<xs:element name="Description">`
	`<xs:complexType>`
	`<xs:choice maxOccurs="unbounded">`
	`<xs:element name="Section" type="xs:string"/>`
	`<xs:element name="Listing">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element name="ListItem" type="xs:string" maxOccurs="unbounded"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`
	`<xs:element ref="DocLink"/>`
	`</xs:choice>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<Description>`
	`<Section>`**This message is used to test the coding or decoding of Information Elements which use the CSN1_S1, CSN1_SHL and S_PADDING coding types.**`</Section>`
	`<Section>`**Reference :**`</Section>`
	`</Description>`

ConstantsSection DocInfoSection PragmasSection Substitut Section

2.1.1.1.1 Listing element

The Listing element itself consists of one to many child elements: ListItem. Each ListItem element can holds any combination of text or digits. The ListItems are intended for conversion of the old document format (MS Word) to the XML document format. For every list that could be identified within descriptive text in the old document format (MS Word) a Listing element has to be created. The content of each list item has then to be transferred to the corresponding child element. The SAPE editor does not provide these elements.

2.1.1.1.2 Linked Description Elements - DocLink

This sub element provides a mechanism to link arbitrary files which are entirely informational, i.e. the tool chain does not use it. Therefore, it is not mandatory in any section. The "Linked Description Elements are intended for additional information, documentation etc.

The SAPE editor offers a table for this sub element. There are two columns:

- The Linked document column,
  that is intended to carry the file name.
- The Type column,
  that provides a pull down list with several suggested file types. Currently SAPE 0.2.5 supports the following standard types: BMP, DOC, GIF, HTML, JPEG, TIFF, TXT. If another file format is desired then the type information should be set to OTHER. The Linked document column should carry the whole file information: <**name**>.<**type**>

Comment [K1]: As well it should be possible to choose any other type. In this case the user has to select the type "OTHER" in this column and in the **Linked document** column in the same row should carry the whole file information: <**name**>.<**type**>

Element DocLink	
Children	**Name** **Comment**
Used by	Element **Description**
Attributes	Name: DocType; Type: xs:string; Use: required; Default; Fixed
XML schema	<xs:element name="DocLink"> <xs:complexType> <xs:sequence> <xs:element ref="Name"/> <xs:element name="Comment" type="xs:string"/> </xs:sequence> <xs:attribute name="DocType" use="required"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="BMP"/> <xs:enumeration value="DOC"/> <xs:enumeration value="GIF"/> <xs:enumeration value="HTML"/> <xs:enumeration value="JPEG"/> <xs:enumeration value="TIFF"/> <xs:enumeration value="TXT"/> <xs:enumeration value="OTHER"/> </xs:restriction> </xs:simpleType> </xs:attribute> </xs:complexType> </xs:element>
XML example	<DocLink DocType="**OTHER**"> <Name>**gmm_service_states.vsd**</Name> <Comment>**GMM State diagram (VISIO 2000)**</Comment> </DocLink>

2.1.1.2 History

The History subsection offers a table containing a manually updated list of changes. It is entirely informational, i.e. the tool chain does not use this section. Nevertheless it is mandatory to enable a detailed change tracking.

Figure 9: SAPE History Table

The developer working on the document is responsible for updating the history list. The initial history data set entry is concatenated with the action "*Add new element*" and should be accomplished by the user.

The SAPE editor provides a table with three columns:

- The Date column,
  that is intended to carry the date of the changes. The data set of format “yyyy-mm-dd” will be automatically generated as soon as new history entry is added.
- The Author column,
  that provides a plain text field. The developer should use his/her own well-known token.
- The Comment column
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory and therefore a preselection is suggested.

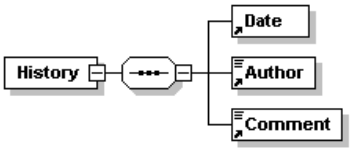


Figure 10: Model of a History Element

Element History	
Children	**Date, Author, Comment**
Used by	Elements **ConstantsSection PragmasSection SubstitutesSection Values Message MsgBasicElem MsgStructElem Function Primitive PrimStructElem PrimBasicElem**

XML schema	`<xs:element name="History">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="Date"/>`
	`<xs:element name="Author" type="xs:string"/>`
	`<xs:element name="Comment" type="xs:string"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<History>`
	`<Date Day="5" Month="12" Year="2001" />`
	`<Author>LG</Author>`
	`<Comment>Initial</Comment>`
	`</History>`

2.1.1.2.1 Document History

This subsection is a modified version of the common History subsection.

Document Historys

	Date	Author	Comment	Year	Number	State
▶	2003-10-31		Initial			BEING_PROCESSED

Figure 11: SAPE Document History

There are three additional columns in the provided table of information:

- The Year column
  is related to the document version and should used to a two-digit number representing the year of change. But any alphanumerical data are possible.
- The Number column
  is related to the document version, too and should used to a three-digit counter number. But any alphanumerical data are possible.
- The State column
  is intended to handle the document status. This information is required and there are only three

values allowed: {*BEING_PROCESSED, SUBMITTED, APPROVED*}. In the XML schema description there exists a sub element called DocStatus with the mandatory attribute State which is associated with the SAPE State column.

The columns Year and Number belong to the sub element DocVersion. The XML schema description handles both information items as mandatory attributes from type *string*.

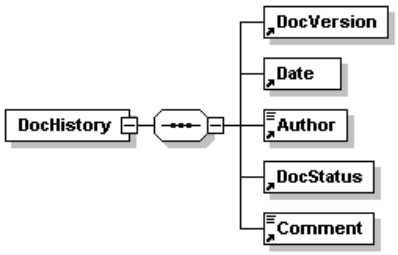


Figure 12: Model of a Document History Element

Element DocHistory	
Children	**DocVersion Date Author DocStatus Comment**
Used by	Element **DocInfoSection**

XML schema	`<xs:element name="DocHistory">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="DocVersion"/>`
	`<xs:element ref="Date"/>`
	`<xs:element name="Author" type="xs:string"/>`
	`<xs:element ref="DocStatus"/>`
	`<xs:element name="Comment" type="xs:string"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<DocHistory>`
	`<DocVersion Number="003" Year="00" />`
	`<Date Day="1" Month="10" Year="2002" />`
	`<Author>CKR</Author>`
	`<DocStatus State="BEING_PROCESSED" />`
	`<Comment>HSCs changes integrated</Comment>`
	`</DocHistory>`

2.1.2 Special Subsections

The items belonging to the special subsection will be used for both types of documents (either SAP or AIM). Unlike the sharable subsection items the usage of these common elements is restricted to the top level in the XML structure. Some sections may be left out. Each section contains a number of subsections, which act as keywords, separating different types of information.

2.1.2.1 Document Information Section

The Document Information Section is a mandatory part although it is entirely informational, i.e. the TI tool chain does not use this section. This section contains global information about the document itself. It serves as a sort of container to group all document relevant information. There is only one *DocInfoSection* element per document.

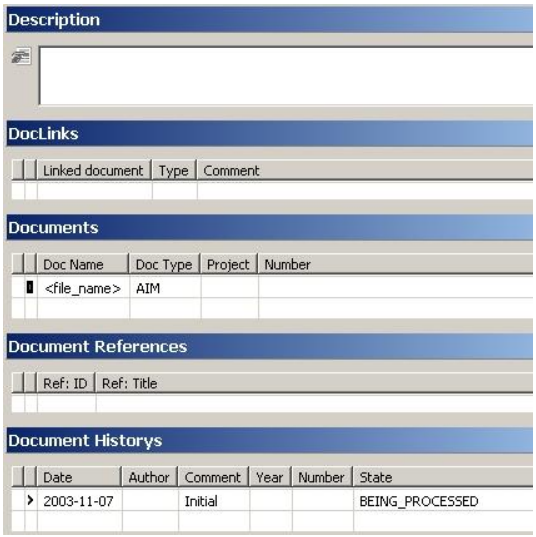


Figure 13: SAPE Document Information Section

The Document Information Section provided by SAPE contains five different parts:

- The Description part should serve as textual explanation of the purpose and meaning of the document itself and corresponds to the child Sectionof the Description element. This *Description* element should be a common introduction to the following document parts. It is strongly recommended to elaborate these descriptions carefully because this will be the only documentation of the primitives or messages specifications as the single source concept agreed.

  The specification engineer should provide enough information to clearly outline the documents interrelations and the intention why this document has been written and how to use it. This important section determines the quality of the documentation targeted at the readers of the XML document or associated to the automatically generated HTML documentation. This part does not appear in any source output from the processed document.

- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.

- The third part is a table to take general, formal document information into account. This table labelled with *Documents* carries one single row, only. The columns correspond to the mandatory XML elements DocName and DocNum.

  The DocName element allows alphanumerical data and should hold the name of a SAP or AIM document without any file extension. This name should follow the rules for SAP and AIM document names.

  The SAPE tool provides in the *Document* table an additional column to set the *Document type*. The XML schema definition handles this type information by the mandatory attribute DocType, which belongs to the mandatory DocName element. This attribute may take one of the possible values *AIM* or *SAP*. The DocType attribute indicates whether an Air Interface Message or a SAP description XML documents is created. By creating a new minimal SAP/AIM document SAPE

extracts the DocType automatically and preselects the filename of the document itself as DocName. The SAPE GUI doesn't allow any DocType changes.

The remaining two columns Project and Number are associated to the DocNum. element. Each is handled by a mandatory attribute. Although these attributes can be chosen without restriction of any kind it is recommended to use a four-digit project number and a three-digit document number dedicated within the project documentation. This document number could be extracted from an entry in the Document History section

- A number of optionally DocRef. elements could be present. These elements are used to define a reference to an external document, which contents might be helpful for understanding. They are handled in the *Document References* table. A DocRef element consists of the mandatory child elements: RefId. and RefTitle. . The column labelled *Ref: ID* belongs to the element RefId and holds an identifier to refer to the reference throughout the document. The second column holds the full title of the document that is referenced and is associated to the RefTitle element.

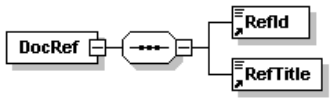


Figure 14: Model of a Document Reference Element

- The History table is intended to track changes. Each row in this table corresponds to one History element. One to many Document History elements (short: *DocHistory*) are allowed. The amount of DocHistory elements will increase by each document change. At least one associated XML element is mandatory and therefore a preselection is suggested.

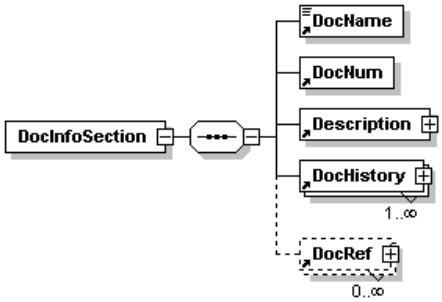


Figure 15: Model of the Document Information Section

Element DocInfoSection	
Children	DocName DocNum Description DocHistory DocRef
Used by	Elements AIM SAP
XML schema	<xs:element name="DocInfoSection"> <xs:complexType> <xs:sequence> <xs:element ref="DocName"/> <xs:element ref="DocNum"/> <xs:element ref="Description"/> <xs:element ref="DocHistory" maxOccurs="unbounded"/> <xs:element ref="DocRef" minOccurs="0" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<DocInfoSection> <DocName DocType="**AIM**">**gmm**</DocName> <DocNum Number="**603**" Project="**8441**" /> <Description> ... </Description> <DocHistory > ... </DocHistory > ... <DocHistory > ... </DocHistory > <DocRef> ... </DocRef > ... <DocRef> ... </DocRef > </DocInfoSection>

2.1.2.2 Pragmas Section

The Pragmas Section serves as a sort of container to group all Pragma declarations. This section is optional. It is used to control and to modify the behaviour of the TI tool chain, e.g. to define a prefix for identifiers (constants, values, etc). There are predefined Pragma keywords that can be assigned a certain value.

Figure 16: SAPE Pragma Section

The Pragmas Section provided by SAPE contains four different parts:

- The Description part should serve as additional information about the Pragmas Section and corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. . This table is associated with the sub element Linked Description Elements belonging to the parent element Description.

- The third part is a table to take one to many Pragma elements (mandatory) into account.
- The History table is intended to track changes. Each row in this table corresponds to one History element. At least one associated XML element is mandatory and therefore a preselection is suggested.

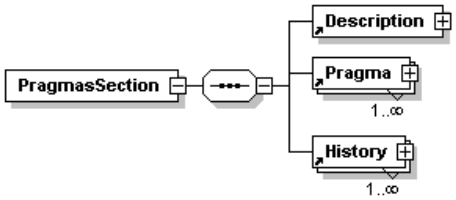


Figure 17: Model of the Pragma Section

Element PragmasSection	
Children	**Description Pragma History**
Used by	Elements **AIM SAP**

XML schema	`<xs:element name="PragmasSection">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="Description"/>`
	`<xs:element ref="Pragma" maxOccurs="unbounded"/>`
	`<xs:element ref="History" maxOccurs="unbounded"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<PragmasSection>`
	`<Description> ... </Description>`
	`<Pragma> ... </Pragma>`
	`...`
	`<Pragma> ... </Pragma>`
	`<History > ... </History >`
	`...`
	`<History > ... </History >`
	`</PragmasSection>`

2.1.2.2.1 Pragma

In the Pragma table one to many items may be present. Each row of the Pragma table represents a separate Pragma declaration itself.

Figure 18: SAPE Pragmas Table

Currently the following pragmas are supported:

PREFIX

COMPATIBILITY_DEFINES

ALWAYS_ENUM_IN_VAL_FILE

ENABLE_GROUP.

CAPITALIZE_TYPENAMES

Each of these will be explained in the following sections.

The SAPE editor provides three columns for mandatory entries in this table:

- The Name column
  provides a plain text field, which is intended to identify an established action.
- The Value column
  supports values facet-valid with respect to enumeration [DEC, BIN, HEX, OCT, ALPHA].
- The Comment column
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory and therefore a preselection is suggested.

Each column relates to an XML element of the same name.

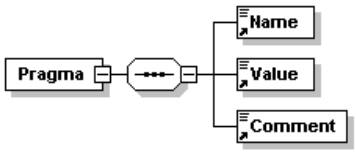


Figure 19: Model of a Pragma Element

Element Pragma	
Children	**Name Value Comment**
Used by	Element **PragmasSection**
XML schema	<xs:element name="Pragma"> <xs:complexType> <xs:sequence> <xs:element ref="Name"/> <xs:element ref="Value"/> <xs:element name="Comment" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<Pragma> <Name>**PREFIX**</Name> <Value ValueType="**ALPHA**">**DL**</Value> <Comment>**Prefix for this document**</Comment> </Pragma>

PREFIX

The pragma **PREFIX** allows all constants, elements and types generated from the SAP document to be automatically prefixed with a letter combination contained in the V**alue** column. The letter combination should follow the TI coding standard, which currently states that SAP content should be prefixed by SAP identifier for the entity to which it belongs (e.g. "RRC"). A special value (**NONE**) can be used to indicate that no prefixing is to be done for the content of the SAP. **Prefixing never applies to primitive names or function names.**

COMPATIBILITY_DEFINES

The pragma **COMPATIBILITY_DEFINES** makes the tool chain generate C pre-processor directives, redefining legacy style declarations to the current standard. The values can be **YES** and **NO** indicating whether to generate them or not.

The combination of **PREFIX** = **NONE** and **COMPATIBILITY_DEFINES** = **YES** is undefined and hence useless, as no prefixing means that all names used in the SAP will remain as is.

ALLWAYS_ENUM_IN_VAL_FILE

The pragma **ALLWAYS_ENUM_IN_VAL_FILE** will make the tool generate an enum for each U8, S8, U16, S16, U32 and S32 type. Each enum containing the constant associated with the corresponding type. The values can be **YES** or **NO**.

If pragma **ALLWAYS_ENUM_IN_VAL_FILE** have a value different from **YES** or is not present, then #define will be generated for such constants.

ENABLE_GROUP

The pragma **ENABLE_GROUP** is used to enable groups. Groups are used for supporting more than one coding standard. The values can be **YES** or **NO**.

If pragma **ENABLE_GROUP** has a value different from **YES** or is not present, then G**roup** columns are ignored. If pragma **ENABLE_GROUP** have the value **YES**, then **Group** columns are mandatory when applicable (see individual table description), and the group cell must contain a value, which may be the special value **none** in which case no entry in the output file is generated for that row. The special group "**none"** is not allowed for types or constants used by a type having another group value than **none**.

When using **Group** columns the output h-files are named according to the group names. That is, the original SAP name does not affect which output files a type is generated in. The group name is used for prefix generation as well (pragma **PREFIX** is ignored if **Group** columns are present). Using **Groups** causes the output to be slightly altered..

CAPITALIZE_TYPENAMES

This pragma is used to indicate whether the generated type names will be capitalized or not. That is for instance whether the generated type for an element called my_u8 would be T_my_u8 or T_MY_U8.

The values can be **YES** or **NO**

2.1.2.3 Constants Section

The Constants Section itself is an optional part and contains information about global constants used in the document. It serves as a sort of container to group all Constant declarations.

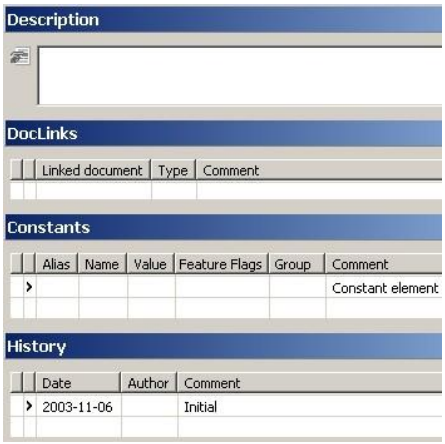


Figure 20: SAPE Constants Section

This section provided by SAPE contains four different parts:

- The Description part should serve as additional information about the Constants Section and corresponds to the child Section of the Description element.

- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The third part is a table to take one to many Constant elements (mandatory) into account.
- The History table is intended to track changes. Each row in this table corresponds to one History element. At least one associated XML element is mandatory and therefore a preselection is suggested.

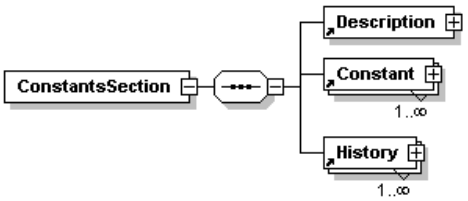


Figure 21: Model of the Constants Section

Element ConstantsSection	
Children	**Description Constant History**
Used by	Elements **AIM SAP**

XML schema	`<xs:element name="ConstantsSection">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="Description"/>`
	`<xs:element ref="Constant" maxOccurs="unbounded"/>`
	`<xs:element ref="History" maxOccurs="unbounded"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<ConstantsSection>`
	`<Description> . . . </Description>`
	`<Constant> . . . </Constant>`
	`. . .`
	`<Constant> . . . </Constant>`
	`<History > . . . </History >`
	`. . .`
	`<History > . . . </History >`
	`</ConstantsSection>`

2.1.2.3.1 Constant

Constants are used as an alias for a user specific value. Constants defined in a document could be used within the document or during the implementation phase to assign a value to a variable. While passing the TI tool chain each Constant element causes a **#define** directive in the C header file, which will be used to give a meaningful name to a constant:

#define *identifier token-string*

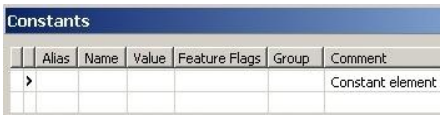


Figure 22: SAPE Constants Table

The Constant element consists of the following child elements:

- The Alias element (mandatory)
  enables any combination of text or digits to identify a user specific value. The SAPE GUI offers

a separate column to support text input. The Alias element acts as identifier for the **#define** directive in the generated C header file. The **#define** directive substitutes *token-string* for all subsequent occurrences of an *identifier* in the source file.

- To define the user specific values (*token-string)* either the Name or the Value column should used.

  Comment [K2]: should be linked item!!!

  The ItemLink element in the XML description represents a reference to an item defined elsewhere in the same or an external document. The SAPE Name column is associated to a child of the ItemLink element and offers the possibility to join a linked item by reference from any constant in the same or in an external document. The SAPE editor provides the possibility to select a new linked element from the Select Repository Entry and to jump to the linked element. The name of the referenced element should be substituted; Alias is the new name of this element.

  The Value element supports locally defined values facet-valid with respect to enumeration [DEC, BIN, HEX, OCT, ALPHA].

- The optional XML Version element is associated to a column named *FeatureFlags*.
  The SAPE editor accepts any combination of text or digits that represents the dependency from feature flags for a specific item. The entries are optional but must comply with coding rules for feature flags.
- The Group column relates to the optional XML element of the same identifier.
  This element could be present to declare a group where the constant belongs. Groups can be used to force the generators of the tool chain to separate the definitions of elements into different output files. Any combinations of text or digits are allowed.
- The Comment column
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory and therefore a preselection is suggested.

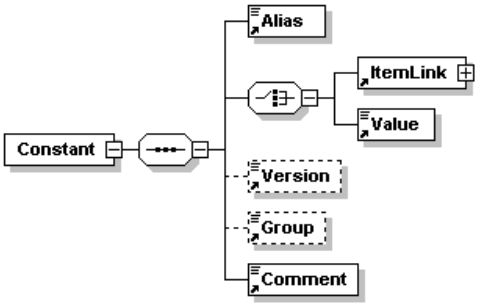


Figure 23: Model of a Constant Element

Element Constant	
Children	**Alias ItemLink Value Version Group Comment**
Used by	Element **ConstantsSection**
XML schema	<xs:element name="Constant"> <xs:complexType> <xs:sequence> <xs:element ref="Alias"/> <xs:choice> <xs:element ref="ItemLink"/> <xs:element ref="Value"/> </xs:choice> <xs:element name="Version" type="xs:string" minOccurs="0"/> <xs:element name="Group" type="xs:string" minOccurs="0"/> <xs:element name="Comment" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<Constant> <Alias>**L3MAX**</Alias> <Value ValueType="**DEC**">**251**</Value> <Comment>**maximum size of a L3 buffer**</Comment> </Constant>

2.1.2.4 Substitutes Section

The Substitutes Section itself is an optional part and serves as a sort of container to group all Substitute declarations.

Figure 24: SAPE Substitutes Section

This section provided by SAPE contains four different parts:

- The Description part should serve as additional information about the Substitutes Section and corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*.. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The third part is a table to take one to many Substitute elements (mandatory) into account.

- The History table is intended to track changes. Each row in this table corresponds to one History element. At least one associated XML element is mandatory and therefore a preselection is suggested.

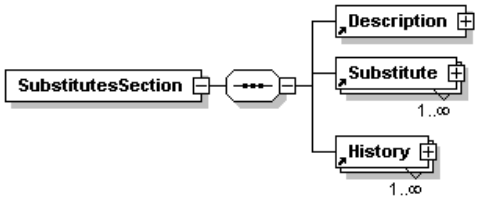


Figure 25: Model of the Substitutes Section

Element SubstitutesSection	
Children	**Description Substitute History**
Used by	Elements **AIM SAP**
XML schema	<xs:element name="SubstitutesSection"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="Substitute" maxOccurs="unbounded"/> <xs:element ref="History" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<SubstitutesSection> <Description> . . . </Description> <Substitute > . . . </Substitute > . . . <Substitute > . . . </Substitute > <History > . . . </History> . . . <History > . . . </History> </SubstitutesSection>

2.1.2.4.1 Substitute

A Substitute is used to define a new name for an existing element. Usually referenced elements could be renamed with an alias name when they are used. Some applications exist, where elements should be renamed without using them, but to define just a new name. This could be useful to support a certain include hierarchy, with the output files generated by the tool chain.

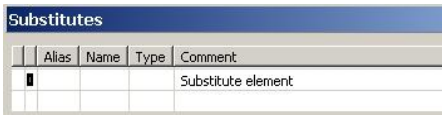


Figure 26: SAPE Substitutes Table

A Substitute is a dedicated element for the case of a substitution and consists of the mandatory child elements:

- The Alias element (mandatory)
  enables any combination of text or digits to identify a user specific value. The SAPE GUI offers a separate column to support text input.
- The ItemLink element in the XML description represents a reference to an item defined elsewhere in the same or an external document. The SAPE Name column is associated to a child of the ItemLink element and offers the possibility to join a linked item by reference from any

member in the same or in an external document. The SAPE editor provides the possibility to select a new linked element from the Select Repository Entry and to jump to the linked element. The name of the referenced element should be substituted; Alias is the new name of this element.

- The Comment column
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory and therefore a preselection is suggested.

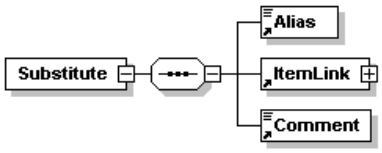


Figure 27: Model of a Substitute Element

The SAPE GUI shows an additional column in the Substitutes table labelled *Type*. The fields of this column can't be edited; they are filled automatically if a substitute belongs to a Structured Message Element.

Element Substitute	
Children	**Alias ItemLink Comment**
Used by	Element **SubstitutesSection**

XML schema	`<xs:element name="Substitute">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="Alias"/>`
	`<xs:element ref="ItemLink"/>`
	`<xs:element name="Comment" type="xs:string"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<Substitute>`
	`<Alias>`**P-TMSI**`</Alias>`
	`<ItemLink>`
	`<DocName DocType="`**AIM**`">`**gmm**`</DocName>`
	`<Name>`**tmsi**`</Name>`
	`</ItemLink>`
	`<Comment>`**Substitute element**`</Comment>`
	`</Substitute>`

2.1.2.5 Values Section

Some specifications require values that must be associated with particular elements. These values may define legal ranges, identified constant values or enumerations that are specific to this element. In AIM descriptions the usage of Values is only allowed within the Basic Message Elements. Respectively, in SAP descriptions the usage of Values is only allowed within Basic Primitive Elements. If a basic element has values, they must be declared in the Values Section. A basic element description contains only a ValuesLink table supporting one to many *Values Link* child elements which could be attached to the basic element itself. The Values Section serves as a sort of container to group all Value declarations.

The Values Section serves as a sort of container to group all Value declarations.

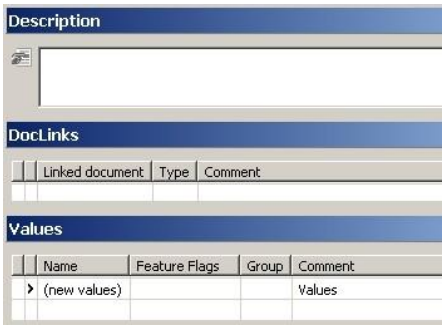


Figure 28: SAPE Values Section

The Values Section provided by SAPE contains three different parts:

- The Description part should serve as additional information about the Values Section and corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The third part is a table to take one to many Values elements (mandatory) into account.

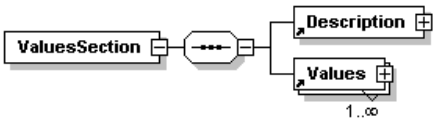


Figure 29: Model of the Values Section

Element ValuesSection	
Children	**Description Values**
Used by	Elements **AIM SAP**
XML schema	<xs:element name="ValuesSection"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="Values" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<ValuesSection> <Description> . . . </Description> <Values> . . . </Values> . . . <Values> . . . </Values> </ ValuesSection >

2.1.2.5.1 Values

Values are constants that can be used as aliases for user specific values. The values are dedicated to the item that they are referenced by and could be used during the implementation phase of a protocol stack entity to assign this value. Besides other tools (e.g. for testing) could use these aliases to display the contents of certain elements in a more readable manner.

The SAPE editor provides a set of tables needed to describe a single value sufficiently. The XML *Values* element serves as a sort of container to group a set of values, that belongs together. Each table relates to an XML element. Table labels and element names can be implicated easily.

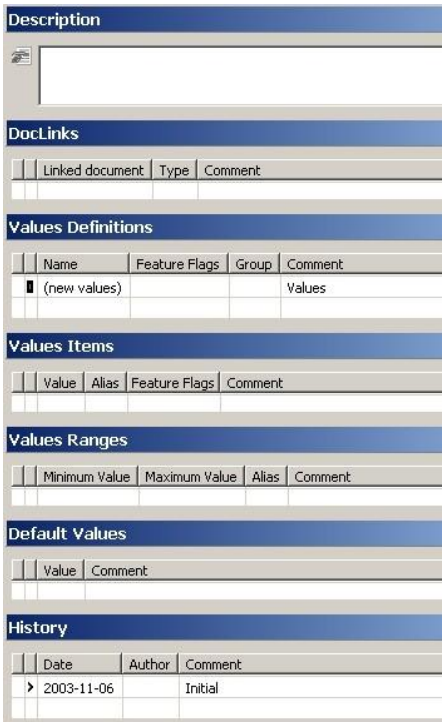


Figure 30: SAPE Values Table

The Values format should correspond to other key elements in the document. Therefore also Values elements have the mandatory elements *Description element* and *History element*. The SAPE GUI offers two parts to take these elements into account:

- The Description part should serve as additional information about the Values and corresponds to the child Section of the Description element.
- The History table is intended to track changes. Each row in this table corresponds to one History element. At least one associated XML element is mandatory and therefore a preselection is suggested.

The Values Definitions table belongs to another mandatory XML element: ValuesDef and acts as a definition element. While passing the TI tool chain each Value element causes a **#define** directive in the C header file. The ValuesDefinitions determine how these values are bound to C identifiers.

The other three tables provided by the SAPE GUI relate to optional XML elements.

- The Values Items table concerns the ValuesDef elements. If the value element identifies an enumeration all possibilities are contained in the Values Items table
- The Values Ranges table is associated with ValuesRange elements. The main purpose of the range specifications is to allow range checking in the test tools used within the TI tool chain.
- The Default Values table involves ValuesDef elements. If the value element does not identify a single value (case of range specification or enumerations) it is possible to choose a default value. This default value should be used if there are not any other assignments.

Each of these child elements consists of nested elements itself.

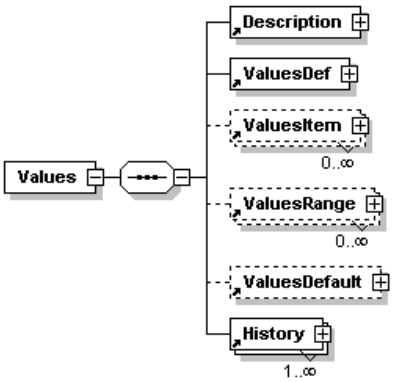


Figure 31: Model of a Value Element

Element Values	
Children	**Description ValuesDef ValuesItem ValuesRange ValuesDefault History**
Used by	Element **ValuesSection**

XML schema	`<xs:element name="Values">`
	`  <xs:complexType>`
	`  <xs:sequence>`
	`  <xs:element ref="Description"/>`
	`  <xs:element ref="ValuesDef"/>`
	`  <xs:element ref="ValuesItem" minOccurs="0" maxOccurs="unbounded"/>`
	`  <xs:element ref="ValuesRange" minOccurs="0" maxOccurs="unbounded"/>`
	`  <xs:element ref="ValuesDefault" minOccurs="0"/>`
	`  <xs:element ref="History" maxOccurs="unbounded"/>`
	`  </xs:sequence>`
	`  </xs:complexType>`
	`</xs:element>`

XML example	`<Values>`
	`  <Description> ... </Description>`
	`  <ValuesDef>`
	`  <Name>VAL_cause_value</Name>`
	`  <Comment>values for cause_value</Comment>`
	`  </ValuesDef>`
	`  <ValuesItem> ... </ValuesItem>`
	`  ...`
	`  <ValuesItem> ... </ValuesItem>`
	`  <ValuesRange> ... </ValuesRange>`
	`  ...`
	`  <ValuesRange> ... </ValuesRange>`
	`  <ValuesDefault> ... </ValuesDefault>`
	`  <History> ... </History>`
	`  ...`
	`  <History> ... </History>`
	`</Values>`

2.1.2.5.1.1 ValuesDef

The Values Definitions table (short: ValuesDef element) provided by SAPE consists of four columns:

- The Name column,
  which entries are mandatory, could be used to reference this value by other elements. This element acts as identifier for the **#define** directive in the generated C header file
  #define *identifier token-string*

- The Feature Flags column
  relates to the *Version* element which offers alphanumerical data fields and represents the dependency from feature flags for a specific item. The entries are optional but must comply with coding rules for feature flags.
- The Group column
  offers alphanumerical data field, too. These optional entries can hold the name of a group. Groups can be used to force the generators of the tool chain to separate the definitions of elements into different output files.
- The Comment column
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory and therefore a preselection is suggested.

Each column relates to an XML element of the same name.

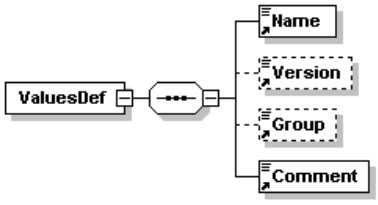


Figure 32: Model of a Values Definition Element

Element ValuesDef	
Children	**Name Version Group Comment**
Used by	Element **Values**

XML schema	`<xs:element name="ValuesDef">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="Name"/>`
	`<xs:element name="Version" type="xs:string" minOccurs="0"/>`
	`<xs:element name="Group" type="xs:string" minOccurs="0"/>`
	`<xs:element name="Comment" type="xs:string"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<ValuesDef>`
	`<Name>VAL_cause_value</Name>`
	`<Comment>values for cause_value</Comment>`
	`</ValuesDef>`

2.1.2.5.1.2 ValuesItem

Each line in the Values Items table (short: ValuesItem element) declares one value and has the child elements:

Comment [K3]: Which is the corresponding child elements of *ValueItem* to transfer the C-macros ?

- The Value column (mandatory)
  supports locally defined values facet-valid with respect to enumeration [DEC, BIN, HEX, OCT, ALPHA].
- The Alias element (mandatory)
  enables any combination of text or digits to identify a user specific value. The SAPE GUI offers a separate column to support text input.
- The Feature Flags column
  relates to the *Version* element which offers alphanumerical data fields and represents the dependency from feature flags for a specific item. The entries are optional but must comply with coding rules for feature flags.

- The Comment column
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory.

Each column relates to an XML element of the same name.

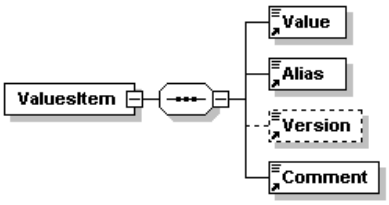


Figure 33: Model of a Values Item Element

Element ValuesItem	
Children	**Value Alias Version Comment**
Used by	Element **Values**

XML schema	`<xs:element name="ValuesItem">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="Value"/>`
	`<xs:element ref="Alias"/>`
	`<xs:element name="Version" type="xs:string" minOccurs="0"/>`
	`<xs:element name="Comment" type="xs:string"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<ValuesItem>`
	`<Value ValueType="DEC">2</Value>`
	`<Alias>ERRCS_IMSI_UNKNOWN</Alias>`
	`<Comment>IMSI unknown in HLR</Comment>`
	`</ValuesItem>`

2.1.2.5.1.3 ValuesRange

In addition to single value definitions, it is possible to declare ranges of values. To display a more meaningful output if no appropriate single values are defined for an element the test tools will use ranges.

To declare a range of values, the Values Ranges table (short ValuesRange element) will be used. The optional ValuesRange element consists of the elements:

- The MinValue element (mandatory),
  which comes from the Minimum Value column, holds alphanumerical data depending on type of value. This element defines the lower boundary value of a value range.
- The MaxValue element (mandatory),
  which comes from the Maximum Value column, holds alphanumerical data depending on type of value. This element defines the upper boundary value of a value range.
- The Alias element (optional)
  enables any combination of text or digits to identify a user specific value. The SAPE GUI offers a separate column to support text input.
- The Comment column
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory.

Similar to the Value element of ValuesDef, the element ValuesRange has an attribute, which defines the type of the values contained in MinValue and MaxValue. Only locally defined values facet-valid with respect to enumeration [DEC, BIN, HEX, OCT, ALPHA] are supported.

Each column relates to an XML element. Column labels and element names can be implicated easily.

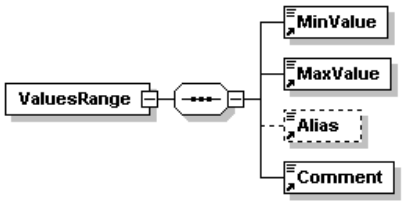


Figure 34: Model of a Values Range Element

Element ValuesRange					
Children	**MinValue MaxValue Alias Comment**				
Used by	Element **Values**				
Attributes	Name	Type	Use	Default	Fixed
	ValueType	valTypeChoice	required		
XML schema	<xs:element name="ValuesRange"> <xs:complexType> <xs:sequence> <xs:element ref="MinValue"/> <xs:element ref="MaxValue"/> <xs:element ref="Alias" minOccurs="0"/> <xs:element name="Comment" type="xs:string"/> </xs:sequence> <xs:attribute name="ValueType" type="valTypeChoice" use="required"/> </xs:complexType> </xs:element>				
XML example	<ValuesRange ValueType="**DEC**"> <MinValue>**48**</MinValue> <MaxValue>**63**</MaxValue> <Comment>**retry upon entry into a new cell** </Comment> </ValuesRange>				

2.1.2.5.1.4 ValuesDefault

Another addition to single value definitions is possible: Besides range declarations defaults can be assigned to values. Similar to the usage of ranges test tools use defaults to display a more meaningful output as a notice if no matching value could be found. The content of the *Comment* column is displayed if no other value is declared.

A default value for an element will be declared using the *Default Values table* providing the following columns:

- The Value column,
  which is only used within air message descriptions for ASN.1 PER encoding rules so far. In such a case, the *Value* element directs CCD to present its content if an optional element of the message was omitted. The presence of these entries is optional
- The Comment column
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory.

Each column relates to an XML element of the same name.

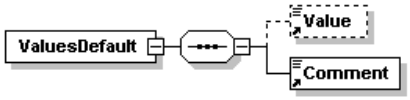


Figure 35: Model of a Default Values Element

Element ValuesDefault	
Children	**Value Comment**
Used by	Element **Values**
XML schema	<xs:element name="ValuesDefault"> <xs:complexType> <xs:sequence> <xs:element ref="Value" minOccurs="0"/> <xs:element name="Comment" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<ValuesDefault> <Value ValueType="**DEC**">**255**</Value> <Comment>**Protocol error, unspecified**</Comment> </ValuesDefault>

2.1.2.6 Annotations Section

The Annotations Section is an optional part that serves as a sort of container. It is entirely informational, i.e. the TI tool chain does not use this section. This section may contain a lot of additional information, which may belong to any element of the document. There is only one Annotations Section element per document.

Primarily this section is intended to offer the SAPE user a convenient possibility to add any comments during the description process. Annotations could be used similar to memos. They allow any kind of hints concerning processing, special sections or elements etc. The concept of annotations supports an easy way to jump from the top level to a certain marked element situated on any nested level.

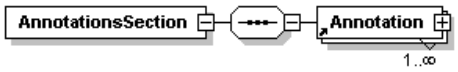


Figure 36: Model of the Annotations Section

SAPE supports the Annotations Section by using the Eclipse standard Task List view. The Task List view displays annotations on resources and enables opening an editor on the resource when the user commands. The task objects are used to mark a location within the document. Adding an annotation creates a task, which appears in the Task view. If the task is selected, the editor may be reopen at the location defined in the Task.

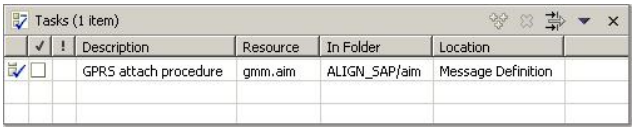


Figure 37: SAPE Annotation Section

The following information is shown in the columns of the Tasks view:

- The left column displays an icon denoting the type of task. Annotations are of the type Task.
- The next column indicates whether the task is completed or not. A task is completed if a check mark is indicated. This parameter is insignificant concerning annotations.

- The Priority column indicates whether the task is of high, normal, or low priority. This column has no relevance although the user can determine the priority of a task.
- The Description column contains a description of the task and is intended to keep the annotation text. The user may edit the description of user-defined tasks by clicking in this column.
- The task view provides two columns to indicate the resource with which the task is associated. The *Resource* column contains the name of the resource and the *In Folder* column indicates the folding information.
- The Location column specifies the element in the associated file where the task marker is located.

Element AnnotationsSection	
Children	**Annotation**
Used by	Elements **AIM SAP**
XML schema	<xs:element name="AnnotationsSection"> <xs:complexType> <xs:sequence> <xs:element ref="Annotation" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<AnnotationsSection> <Annotation> </Annotation> . . . <Annotation> </Annotation> </AnnotationsSection>

2.1.2.6.1 Annotation Element

The Annotation Element is used to keep additional information, which may belong to any element of the document. SAPE provides a dialog box to specify a new annotation (see Figure 38). Once an annotation is added to an arbitrary element a task will be created, which will appears in the Task view.

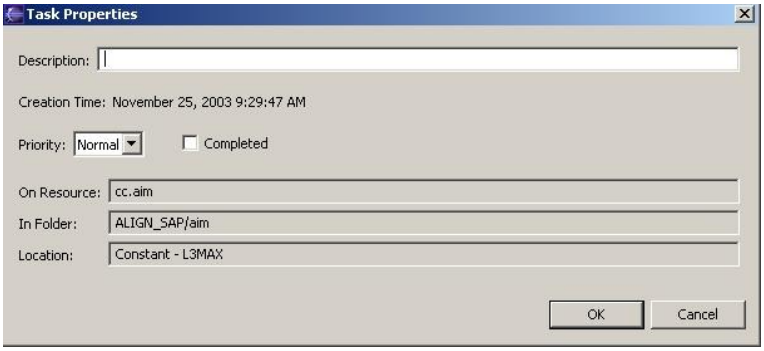


Figure 38: SAPE Dialog Box to Specify a New Annotation

The related XML Annotation Element consists of the following child elements:

- The Data Target refers to a single element within the XML description file which should be associated with this annotation element itself. This child element is mandatory.
- The Comment element enables any combination of text or digits and is mandatory too. This element relates to the Description field in the dialog box above and to the Description column in the Task View respectively. The Comment element is intended to keep the annotation text

- The Priority element is associated with the Priority column, which indicates whether the task is of high, normal, or low priority. This element has no relevance and therefore it is optional.
- The Done element is also optional because the Eclipse standard Task List view requires this parameter although it is insignificant concerning annotations. This element allows indicating whether the task is completed or not.

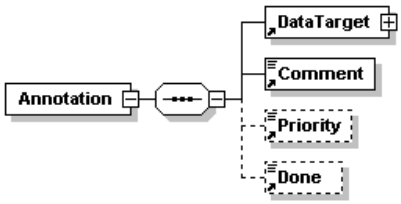


Figure 39: Model of an Annotation Element

Element Annotation	
Children	**DataTarget Comment Priority Done**
Used by	Element **AnnotationsSection**

XML schema	`<xs:element name="Annotation">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="DataTarget"/>`
	`<xs:element name="Comment" type="xs:string"/>`
	`<xs:element name="Priority" type="xs:string" minOccurs="0"/>`
	`<xs:element name="Done" type="xs:string" minOccurs="0"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<Annotation>`
	`<DataTarget>`
	`<ElementOffset>...</ElementOffset>`
	`...`
	`<ElementOffset>...</ElementOffset>`
	`</DataTarget>`
	`<Comment>`**GPRS attach procedure**`</Comment>`
	`<Priority>`**1**`</Priority>`
	`<Done>`**false**`</Done>`
	`</Annotation>`

2.1.2.6.2 Data Target

The Data Target element relates to the Location column, which specifies the element in the associated file where the task marker is located. This element is composed of a set of child elements called ElementOffset. The ElementOffset elements enable any combination of text or digits, but it is recommended to use positive integer values.

Each ElementOffset element characterises a certain level in the XML information tree. The element value indicates the number of a particular node assuming a consecutive numbering beginning with zero (see XML example and its explanation). SAPE uses these ElementOffset elements to determine the element name of the location where the task marker is placed.

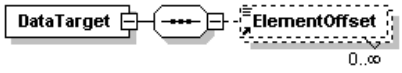


Figure 40: Model of a Data Target Element

Element DataTarget	
Children	**ElementOffset**
Used by	Element **Annotation**
XML schema	<xs:element name="DataTarget"> <xs:complexType> <xs:sequence> <xs:element name="ElementOffset" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<DataTarget> <ElementOffset>**0**</ElementOffset> <ElementOffset>**2**</ElementOffset> <ElementOffset>**1**</ElementOffset> <ElementOffset>**1**</ElementOffset> </DataTarget>
Associated element	<AIM xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance /* level 0, element 0 */ xsi:noNamespaceSchemaLocation="**aim.xsd**"> + <DocInfoSection> + <ConstantsSection> - <MessagesSection> /* level 1, element 2 */ + <Description> - <Message> /* level 2, element 1 */ + <Description> - <MsgDef> /* level 2, element 1 */ <Name>**attach_request**</Name> <MsgID Direction="**UPLINK**" IDType="**DEC**">**1**</MsgID> <Comment>**attach request**</Comment> </MsgDef> + <MsgItem Presentation="**MANDATORY**"> . . . + <MsgItem Presentation="**MANDATORY**"> + <History> </Message> . . . ⇒ **The data target element above concatenates an annotation to the MsgDef element.**

2.1.3 Nontrivial Sub-Elements

Some sub elements, which may occur in different context, require more detailed explanation. These elements are listed here to provide additional information about the data they may contain. This section should serve primarily as a reference for these elements.

2.1.3.1 Alias

The *Alias* element supports alphanumerical data input. This element holds an alias name for a user specific value or to change the name of an existing element. The alias name will typically result in a #define expression, a type name or a variable name according to the C programming language. Therefore the in-house coding rules for C names apply.

Element Alias	
Type	**xs:string**
Used by	Elements **Constant Substitute ValuesItem ValuesRange MsgItem MsgStructElemItem FuncArg PrimItem PrimStructElemDef PrimStructElemItem**
XML schema	<xs:element name="Alias" type="xs:string"/>
XML example	<Alias>**NO_KEY**</Alias>

2.1.3.2 DocName

The DocName elements support any combination of text or digits. This alphanumerical data field is intended to holds the name of a SAP or AIM document without any file extension and should follow the rules for SAP and AIM document names. The file extension is determined by the required attribute *DocType*. Attributes are used to associate name-value pairs with elements. The *DocType* attribute determines how to interpret the DocName element's content. This attribute can be set either to AIM or to SAP.

The SAPE tool recognizes the right *DocType* setting automatically while selecting a new element to link. DocType must be set to SAP or AIM.

Element DocName	
Type	extension of **xs:string**
Used by	Elements **DocInfoSection ItemLink ValuesLink**
Attributes	Name: DocType; Type: docTypeChoice; Use: required; Default: ; Fixed:
XML schema	`<xs:element name="DocName"><xs:complexType><xs:simpleContent><xs:extension base="xs:string"><xs:attribute name="DocType" type="docTypeChoice" use="required"/></xs:extension></xs:simpleContent></xs:complexType></xs:element>`
XML example	`<DocName DocType="AIM">gmm</DocName>`

2.1.3.3 Group

This alphanumerical data element holds the name of a group. *Groups* can be used to force the generators of the tool chain to separate the definitions of elements into different output files.

2.1.3.4 Name

This element enables any combination of text or digits and is intended to hold a name for an element. The name will typically result in a type expression or a variable name according to the C programming language. Therefore the in-house coding rules for C names apply

Element Name	
Type	**xs:string**
Used by	Elements **DocLink ItemLink Pragma ValuesDef ValuesLink MsgBasicElemDef MsgDef MsgStructElemDef UnionTag FuncDef PrimDef PrimStructElemDef PrimBasicElemDef**
XML schema	`<xs:element name="Name" type="xs:string"/>`
XML example	`<Name>b_csn1_ies</Name>`

2.1.3.5 ItemLink

The ItemLink element represents a reference to an item defined elsewhere locally in the same or externally in another document. It was defined by the name of the document and the name of the item.

Note: Links work only in the sequence given by makcdg.mak[2], but not inversely!
Example: grr.aim should be able to use items from rr.aim but not inversely.

[2] makcdg.mak can be found by \g23m\condat\int\bin\makcdg.bat

Therefore this element consists of the mandatory child elements:

- The element DocName holds the name of the document defining the item to link. In case of a local reference the DocName will be the same as the currently in use, otherwise the external document name should be given.
- The Name
  refers to the name of the item to link.

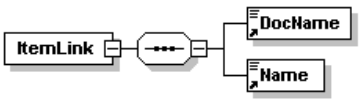


Figure 41: Model of an ItemLink Element

Element ItemLink	
Children	**DocName Name**
Used by	Elements **Constant Substitute MsgItem MsgStructElemItem FuncArg FuncRet PrimItem PrimStructElemItem**

XML schema	`<xs:element name="ItemLink">`
	`  <xs:complexType>`
	`  <xs:sequence>`
	`  <xs:element ref="DocName"/>`
	`  <xs:element ref="Name"/>`
	`  </xs:sequence>`
	`  </xs:complexType>`
	`</xs:element>`

XML example	`<ItemLink>`
	`  <DocName DocType="AIM">gmm</DocName>`
	`  <Name>tmsi</Name>`
	`</ItemLink>`

2.1.3.6 Feature Flags

This element supports feature flags in message and primitive documents. The goal is to adjust the protocol stack in a sensitive way by these feature flags and to allow that the supported messages and primitives are dependent on the configuration used, e.g. in order to save memory for simple configuration.

The existing build process is only able to configure the protocol stack in a coarse way. It is possible to build a pure GSM stack, a stack with GSM and GPRS and other major properties e.g. Fax&Data. But within the major properties there are a lot of fine adjustments

SAPE provides a column named Feature Flags to enable binding feature flags to a constant, variable, value, element, structure, message or primitive. This mechanism admits to switch the bound item on or off depending on enabling/disabling a particular feature flag.

It is possible to use a single feature flag, only. But if more than one feature flag are needed, these feature flags have to be combined to a Boolean expression (e.g. !FF_x AND (!FF_y or FF_z>99)). If an element should be switched off in the whole document, use the feature flag column of the element itself. If it should be switched off only in a given item (e.g. structure), use the feature flag column of the element within this item. See the example below:

Structured Element Definitions

	Name	Alias	Type	Feature Flags	Group	Comment
›	struct_element_2		STRUCT		GRP_5	example struct from sape_example.pdf

Structured Element Items

	Name	Type	Feature Flags	Comment	Presence
›	basic_elem_1	U8	FF_TI_DUAL_MODE	Primitive structure element item	MANDATORY
›	basic_elem_2	U16		Primitive structure element item	MANDATORY
›	basic_elem_3	U32		Primitive structure element item	MANDATORY

Figure 42: Example 1 of SAPE Feature Flag Specification

The following table gives an overview of the valid Boolean expression and their meaning.

Expression	Example	Meaning
<name>	FF_X	Allowed characters in a flag are given by [A-Z0-9_].
!	!FF_X	Invert the logical value of an evaluated sub-expression or a given feature flag
()	(FF_X)	Grouping feature flag items
AND, OR	FF_X AND FF_Y OR FF_Z	Logical Operators
>, >=, <, <=, !=, ==	FF_x > 99	Comparison with given constants

The arithmetical operators "+" , "-" and "=" are not supported.

For the evaluation of the Boolean expression there is a file containing the feature flags definition that provides information which feature flags are defined with which value. The feature flag catalogue (cf. [5.]) shall provide a reference to the use, meaning and correct spelling of the feature flags: it defines the name of a FF and its semantics.

The TI tool chain converts a message or primitive specification document to C header files and generates for the TI Coder Decoder "CCD" the "cdg tables". In the C header files elements of a structure bound with feature flags are wrapped with C pre-processor statements: Feature flags are to be defined by **#define** and **#undef** expressions. **#undef FF_xx** corresponds to **!F_xx** in the Feature Flags column of the SAP/AIM document

If there are defined values to a variable, which are going to symbolic constants **#define <const_name> 0x1234** in the "mconst.cdg" or "pconst.cdg"files, these **#defines ...** remains as they are. When the C pre-processor does not substitute a symbolic constant in the C source, then this symbolic constant does not contribute to the object code.

In the generated header file there are no pre-processor statements around such structure elements which are affected by feature flags. Instead, a structure element for which the feature flag expression is logical FALSE, is given as comment line. And if the FF expression is logical FALSE, the element itself appears as comment too. See the type definition the C header file belonging to Figure 42 below:

```
typedef struct
{
/* #if FF_TI_DUAL_MODE
  U8  basic_elem_1;  */   /*<  0:  0> basic element 1 */
  U16 basic_elem_2;       /*<  0:  2> basic element 2 */
  U32 basic_elem_3;       /*<  2:  4> basic element 3 */
} T_SAPE_EXAMPLE_struct_element_2;
```

In this example it is not clear if the basic_elem_1 is generally not available for all structures or it is only forbidden in the given structure. Both cases have the same print in the header file.

NOTE : It is not recommended but possible to include an external element into the current AIM/SAP specification depending on a particular set feature flags:

```
typedef struct
{
  U8    ch_type;          /*<  0:  1> T_VAL_L2_CHANNEL[3],  Layer 2 channel-type */
/* ELEM-FF: FF_TI_DUAL_MODE
  U8    sapi;  */         /*<  0:  0> T_VAL_SAPI,  service access point identifier */
  U8    sapi;             /*<  1:  1> T_VAL_SAPI,  service access point identifier */
/* ELEM-FF: FF_TI_DUAL_MODE
  T_CAUSE_ps_cause    ps_cause;   */ /*<  0:  0> Cause element containing result of
                                       operation (type defined in "p_8010_153_cause_include.h") */
  U8    cs;                 /*<  2:  1> T_DL_VAL_cs,  error cause */
  U8    _align0;            /*<  3:  1> alignment */
} T_DL_RELEASE_IND;
```

Obviously the commented elements have no impact on the alignment. Here the elements **sapi** and **cause** are not included from p_8010_153_cause_include.sap, since the flag **FF_TI_DUAL_MODE** is not set. This corresponds to **#undef FF_TI_DUAL_MODE** in the feature-flags file and **!FF_TI_DUAL_MODE** in the "Feature Flags" column of the PRIM description. This corresponds to the following description in the SAP/AIM Editor:

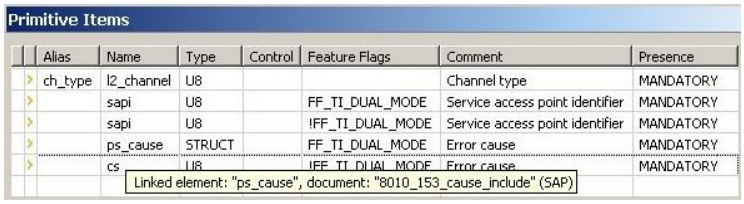


Figure 43: Example 2 of SAPE Feature Flag Specification

Feature flags have an impact on the generation of the "cdg tables", too. The TI tool chain has always to evaluate the feature flag and then to decide whether that element is going to the "cdg tables" or not. Therefore the feature flags affect the size of the "cdg tables" and valuable memory size on the target can be saved. Any object in the intermediate files (*.mdf/pdf files) for which the feature flags expression is logical FALSE will not appear in the tables of CCDDATA. Nor they will have any impact on the other objects, e.g. offset in the C-structures, number of structure elements.

2.1.3.7 UnionTag

Basically the UnionTag element is an alias for a user specific value. It is used as a tag identifier of union elements that indicates which union element of all possible elements is present. The alias name will typically result in an item of an enum expression according to the C programming language. Therefore the in-house coding rules for C names apply.

A UnionTag consists of two child elements:

- The mandatory Name element
  can take any combination of text or digits and is intended to hold the tag name.
- The optional Value element
  supports locally defined values facet-valid with respect to enumeration [DEC, BIN, HEX, OCT, ALPHA].

[3] T_VAL_L2_CHANNEL is an enumeration defined in the appropriate *.val file.

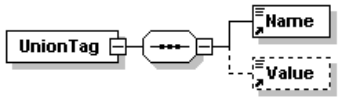


Figure 44: Model of a Union Tag Element

element UnionTag	
Children	Name Value
Used by	Elements MsgStructElemItem PrimStructElemItem
XML schema	<xs:element name="UnionTag"> <xs:complexType> <xs:sequence> <xs:element ref="Name"/> <xs:element ref="Value" minOccurs="0"/> </xs:sequence> </xs:complexType> </xs:element>

2.1.3.8 ValuesLink

The ValuesLink element represents a reference to a set of values defined elsewhere locally in the same or externally in another document. The name of the values set in conjunction with the name of the document defines a reference to a set of values in a sufficient way. Therefore this element consists of the mandatory child elements:

- The element DocName holds the name of the document defining the item to link. In case of a local reference the DocName will be the same as the currently in use, otherwise the external document name should be given.
- The name of the *Value* element that is referred to will be put in the Name element of ValuesLink.

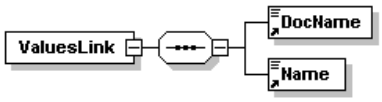


Figure 45: Model of a Value Link Element

Element ValuesLink	
Children	DocName Name
Used by	Elements MsgBasicElem PrimBasicElem
XML schema	<xs:element name="ValuesLink"> <xs:complexType> <xs:sequence> <xs:element ref="DocName"/> <xs:element ref="Name"/> </xs:sequence> </xs:complexType> </xs:element>
XML example [12.] part 10.5.1.4 (mobile identity)	<ValuesLink> <DocName DocType="**AIM**">**gmm**</DocName> <Name>**VAL_type_of_identity**</Name> </ValuesLink>

2.1.4 Trivial Sub-Elements

The intention of these trivial Sub-Elements, which may occur in different context, does not need any further explanation because their names are self-explanatory. They are listed here because of their correct appearance required by the XML schema. The SAPE tool supports the right formatting. But if an XML document shall be written or modified by using a text editor the user has to be informed how the format must look like.

Element Date

Used by: Elements **DocHistory** **History**

Attributes	Name	Type	Use	Default	Fixed
	Day	xs:string	required		
	Month	xs:string	required		
	Year	xs:string	required		

XML schema:

```
<xs:element name="Date">
  <xs:complexType>
    <xs:attribute name="Day" type="xs:string" use="required"/>
    <xs:attribute name="Month" type="xs:string" use="required"/>
    <xs:attribute name="Year" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

XML example:

```
<Date Day="29" Month="7" Year="1999"/>
```

Element DocNum

Used by: Element **DocInfoSection**

Attributes	Name	Type	Use	Default	Fixed
	Project	xs:string	required		
	Number	xs:string	required		

XML schema:

```
<xs:element name="DocNum">
  <xs:complexType>
    <xs:attribute name="Project" type="xs:string" use="required"/>
    <xs:attribute name="Number" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

XML example:

```
<DocNum Number="603" Project="8441" />
```

Element DocRef

Children: **RefId** **RefTitle**

Used by: Element **DocInfoSection**

XML schema:

```
<xs:element name="DocRef">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="RefId" type="xs:string"/>
      <xs:element name="RefTitle" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

XML example:

```
<DocRef>
  <RefId>[1]</RefId>
  <RefTitle>RFC 1661 IETF STD 51 July 1994The Point-to-Point Protocol (PPP)</RefTitle>
</DocRef>
```

Element DocStatus	
Used by	Element **DocHistory**
Attributes	Name: State; Type: xs:string; Use: required; Default: ; Fixed:
XML schema	`<xs:element name="DocStatus"><xs:complexType><xs:attribute name="State" use="required"><xs:simpleType><xs:restriction base="xs:string"><xs:enumeration value="BEING_PROCESSED"/><xs:enumeration value="SUBMITTED"/><xs:enumeration value="APPROVED"/></xs:restriction></xs:simpleType></xs:attribute></xs:complexType></xs:element>`
XML example	`<DocStatus State="BEING_PROCESSED" />`

Element DocVersion	
Used by	Element **DocHistory**
Attributes	Name: Year; Type: xs:string; Use: required; Default: ; Fixed: Name: Number; Type: xs:string; Use: required; Default: ; Fixed:
XML schema	`<xs:element name="DocVersion"><xs:complexType><xs:attribute name="Year" type="xs:string" use="required"/><xs:attribute name="Number" type="xs:string" use="required"/></xs:complexType></xs:element>`
XML example	`<DocVersion Number="004" Year="99" />`

Element Value	
Type	extension of **xs:string**
Used by	Elements **Constant** **Pragma** **UnionTag** **ValuesDefault** **ValuesItem**
Attributes	Name: ValueType; Type: valTypeChoice; Use: required; Default: ; Fixed:
XML schema	`<xs:element name="Value"><xs:complexType><xs:simpleContent><xs:extension base="xs:string"><xs:attribute name="ValueType" type="valTypeChoice" use="required"/></xs:extension></xs:simpleContent></xs:complexType></xs:element>`
XML example	`<Value ValueType="DEC">2</Value>`

2.2 Message Specific Part

This part describes how to write TI Air Interface Message documents. Air interface messages are peer-to-peer messages between a MS and its network peer. These messages are standardized by ETSI/3GPP, and thus have well defined formats. The TI Air Interface Message documents describe how data in air interface messages are organized, and how they can be used in entities. They operate at bit-level as opposed to Service Access Points (SAPs), which normally operate at byte level.

Air Interface Messages are documents written in XML as part of the high-level design phase. When the air interface messages they describe are needed in actual code, the documents are run through the TI tool chain (cf. GTC Generic Tool Chain), which produce header files and other data needed in program code.

An Air Interface Message XML document is created from a XML schema by using the SAPE tool and choosing AIM (air interface message) type when creating a new document. The list below shows the elements belonging to the top level. Some sections may be left out; these sections are marked [optional].

Document Information	Container for all document relevant information
Pragma [optional]	Used to control and to modify the behaviour of the TI tool chain
Constant [optional]	Contains global constants and used to assign a value to a variable
Messages	The actual air interface message descriptions (e.g. ATTACH REQUEST)
Structured Elements [optional]	Elements in air interface messages (e.g. MS class mark)
Basic Elements [optional]	Basic types/values (e.g. MS type 2)
Substitute [optional]	Used to define a new name for an existing element
Values [optional]	Used as aliases for user specific values
Annotations [optional]	Container for additional information belonging to any document's

Each section above contains a number of subsections, which act as keywords, separating different types of information. Some of these subsections occur in both types of documents (either SAP or AIM) and are already mentioned in part 2.1.

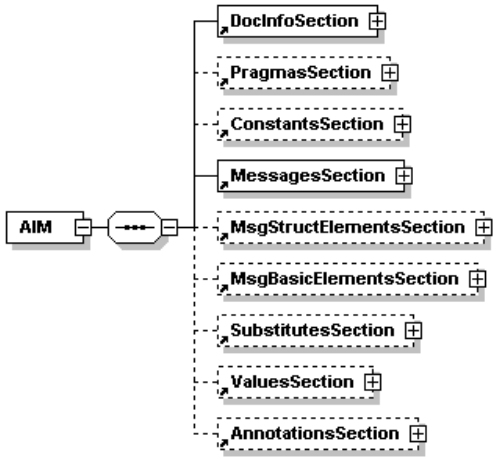


Figure 46: Model of an AIR Messages Description

Element AIM	
Children	DocInfoSection PragmasSection ConstantsSection MessagesSection MsgStructElementsSection MsgBasicElementsSection SubstitutesSection ValuesSection AnnotationsSection
XML schema	<xs:element name="AIM"> <xs:complexType> <xs:sequence> <xs:element ref="DocInfoSection"/> <xs:element ref="PragmasSection" minOccurs="0"/> <xs:element ref="ConstantsSection" minOccurs="0"/> <xs:element ref="MessagesSection"/> <xs:element ref="MsgStructElementsSection" minOccurs="0"/> <xs:element ref="MsgBasicElementsSection" minOccurs="0"/> <xs:element ref="SubstitutesSection" minOccurs="0"/> <xs:element ref="ValuesSection" minOccurs="0"/> <xs:element ref="AnnotationsSection" minOccurs="0"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<AIM xmlns:xsi="**http://www.w3.org/2001/XMLSchema-instance**" xsi:noNamespaceSchemaLocation="**aim.xsd**"> ... PragmasSection ... PragmasSection ... ... SubstitutesSection ... Subs-titutesSection ... AnnotationsSection ... AnnotationsSection

2.2.1 Messages Section

The *Messages Section* deals with the Message Elements supported by the SAPE editor in table format. It handles the *Messages Section* element, which comprises **all** messages declared within an AIM document.

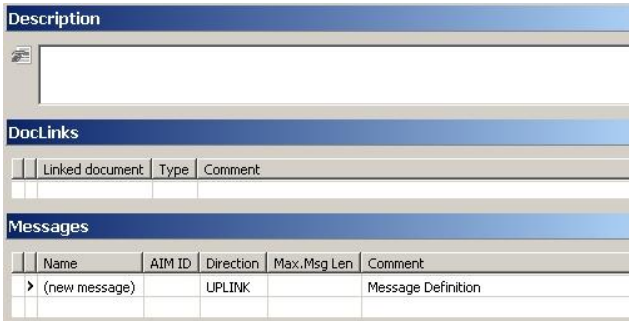


Figure 47: SAPE Messages Section

The *Messages Section* serves as a container for all message elements. The SAPE GUI provides three different parts:

- The Description part should serve as additional information and contains a textual description of the information in the Messages table. The input corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- One to many child elements Message
  which are described in a particular subsection of this document, deal with the internal structure of the Message elements. At least one Message element has to be present. The SAPE editor offers a table labelled *Messages* providing a separate row for each child element.

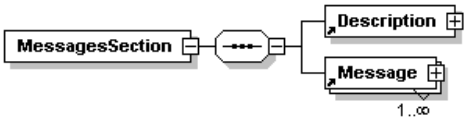


Figure 48: Model of the Messages Section

Element MessagesSection	
Children	**Description** **Message**
Used by	Element **AIM**
XML schema	<xs:element name="MessagesSection"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="Message" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<MessagesSection> ... ... </MessagesSection>

2.2.1.1 Message

The SAPE GUI provides a table labelled *Messages* to hold the *Message* elements, which comprise all instruments being necessary to define a message in an AIM document. The Message element serves as a sort of container itself to group a set of information defining its properties. Each row in the Message table is associated with another set of tables.

Except the DocLinks table each table relates to an XML element. Table labels and element names can be implicated easily.

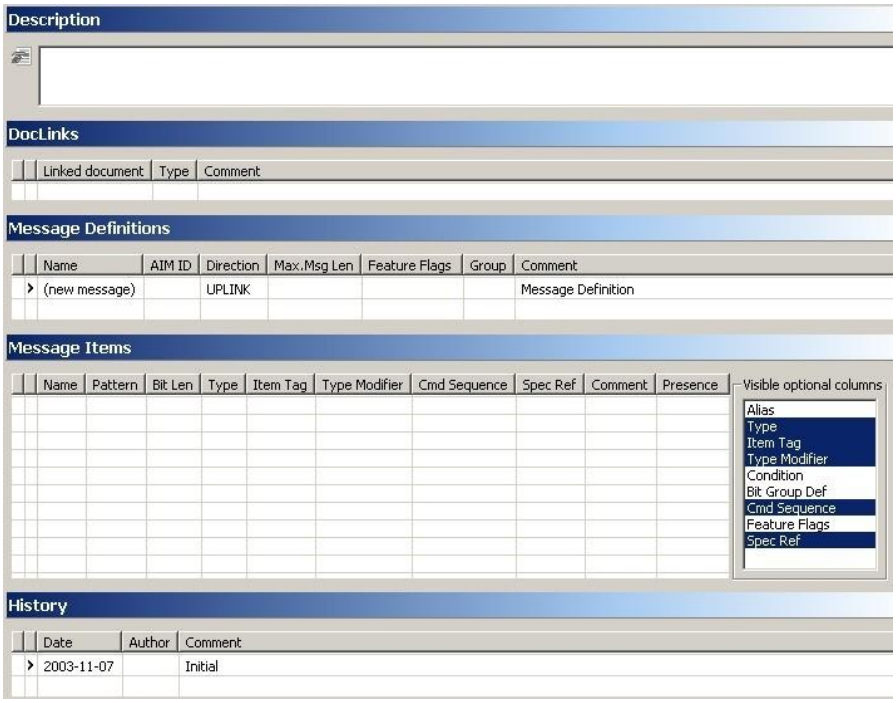


Figure 49: SAPE Message

The Message format should correspond to other key elements in the document. Therefore also Message elements have the mandatory elements *Description element* and *History element*. The SAPE GUI offers three parts to take these elements into account:

- The Description part should serve as additional information about each Message and corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The History table is intended to track changes. Each row in this table corresponds to one History element. At least one associated XML element is mandatory and therefore a preselection is suggested.

The *Message Definitions* table belongs to another mandatory XML element: MsgDef. It acts as a definition element. In case that the layout of the message should be assigned to more than only one message definition, additional MsgDef elements could be present.

The other table provided by the SAPE GUI relate to an optional XML element.

- The *Message Items* table supports one to many MsgItem child elements. These elements are optional and could be attached to each Message element if values should be assigned to.

Each of the last two child elements consists of nested elements itself.

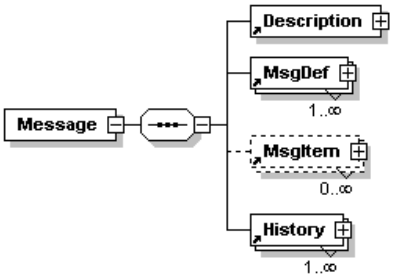


Figure 50: Model of a Message Element

Element Message	
Children	**Description MsgDef MsgItem History**
Used by	Element **MessagesSection**
XML schema	<xs:element name="Message"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="MsgDef" maxOccurs="unbounded"/> <xs:element ref="MsgItem" minOccurs="0" maxOccurs="unbounded"/> <xs:element ref="History" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<Message> <Description>. . . </Description> <MsgDef> /* CSN1 IE msg */ <Name>**b_csn1_ies**</Name> <MsgID Direction="**BOTH**" IDType="**BIN**">**00001000**</MsgID> <Comment>**CSN1 IE msg**</Comment> </MsgDef> <MsgItem Presentation="**MANDATORY**"> /* Message Type */ <ItemLink> <DocName DocType="**AIM**">**xx**</DocName> <Name>**msg_type**</Name> </ItemLink> <Type>**GSM3_V**</Type> <SpecRef>-</SpecRef> <Comment>**Message Type**</Comment> </MsgItem> <MsgItem Presentation="**OPTIONAL**"> /* CSN1_S1 IE */ <ItemLink> <DocName DocType="**AIM**">**xx**</DocName> <Name>**csn1_s1**</Name> </ItemLink> <Type>**GSM4_TLV**</Type> <ItemTag TagType="**HEX**">**18**</ItemTag> <SpecRef>-</SpecRef> <Comment>**CSN1_S1 IE**</Comment> </MsgItem> <MsgItem Presentation="**OPTIONAL**"> /* CSN1_SHL IE */ <ItemLink> <DocName DocType="**AIM**">**xx**</DocName> <Name>**csn1_shl**</Name> </ItemLink> <Type>**CSN1_SHL**</Type> <SpecRef>-</SpecRef> <Comment>**CSN1_SHL IE**</Comment> </MsgItem> <MsgItem Presentation="**OPTIONAL**"> /* S_PADDING */ <Spare> <Pattern>**00101011**</Pattern> <BitLen>**8**</BitLen> </Spare> <Type>**S_PADDING**</Type> <Control> <CmdSequence>**MAX_PADD**</CmdSequence> </Control> <SpecRef>-</SpecRef> <Comment>**Padding**</Comment> </MsgItem> <History> . . . </History> </Message>

Associated GSM specification	Message type: CSN1 IE msg

Information element	Presence
Message Type	M
CSN1_S1 IE	O
CSN1_SHL IE	O
S_PADDING	O

Table 3: Example of a Message (CSN1 IE msg)

2.2.1.1.1 Message Definitions

The *Message Definitions* table provided by the SAPE GUI holds the *MsgDef* element, which defines the key parameters of a *Message Definition* element in an AIM document. Except for the columns labelled *AIM ID* and *Direction* all other columns in the *Message Definitions* table correspond to a child element of the MsgDef element. The column labels should be more self-explanatory than the child element names.

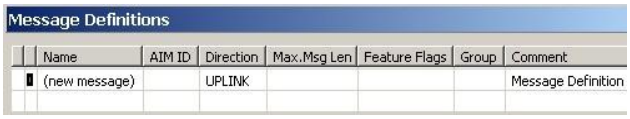


Figure 51: SAPE Message Definition Table

The MsgDef element consists of the following child elements:

- The Name element serves as unique identification.
  This element is mandatory and could be used to reference this *Message Element* by other elements.
- The *AIM ID* column provides the input mask for the mandatory MsgID[4] element and holds the numerical identifier for that message. The message ID has to be separated into message identifier and identifier type. The message identifier will be the contents of *MsgID element,* whereas the identifier type will set the *IDType* attribute accordingly. *MsgID* is a little bit different because the content is splitted into separate elements. The plain identifier number goes into the *MsgID* element, whereas the type of message identifier, which values are facet-valid with respect to enumeration [DEC, BIN, HEX, OCT, ALPHA], is stored in the *IDType* attribute.
- The Max.Msg Len column represents the optional MsgLenMax element.
  With the *MsgLenMax* element the user can manually define the maximum message length in bytes in the real world environment. It is helpful to separate this from the theoretical length of a message, when designing buffers for messages. Although the XML schema description allows any combination of text or digits it is recommended to use values representing numbers only.
- The optional Version element
  accepts any combination of text or digits that represents the dependency from feature flags for the definition of a specific message element. This element relates to the Feature Flags column. The entries are optional but must comply with coding rules for feature flags.
- The Group element (optional)
  could be present to declare a group where the *Message Element* belongs. Groups can be used to force the generators of the tool chain to separate the definitions of elements into different output files. Any combination of text or digits is allowed.
- The Comment element (mandatory)
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory and therefore a preselection is suggested.

[4] The message ID is an absolutely important entry for the coder/decoder.

The SAPE tool provides in the *Message Definitions* table an additional column to set the direction information for each *Message Element*. The XML schema definition handles this direction information by the mandatory attribute *Direction*[5], which belongs to the mandatory MsgID element. This attribute may take one of the possible values UPLINK, DOWNLINK or BOTH. The *Direction* attribute indicates the message flow direction according to the ETSI/3GPP specifications:

UPLINK - An "uplink" is a unidirectional radio link for the transmission of signals from a UE towards a core network.

DOWNLINK - A “downlink” is a unidirectional radio link for the transmission of signals from a core network towards a UE.

BOTH - This direction applies identification to a bidirectional radio link for transmission of signals as well signals from a UE towards core network as vice versa.

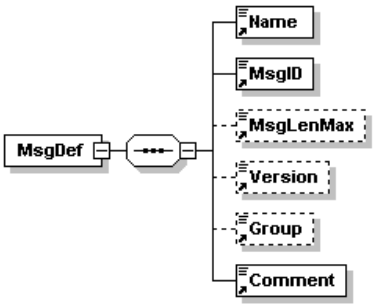


Figure 52: Model of a Message Definition Element

Element MsgDef	
Children	**Name MsgID MsgLenMax Version Group Comment**
Used by	Element **Message**
XML schema	<xs:element name="MsgDef"> <xs:complexType> <xs:sequence> <xs:element ref="Name"/> <xs:element ref="MsgID"/> <xs:element name="MsgLenMax" type="xs:string" minOccurs="0"/> <xs:element name="Version" type="xs:string" minOccurs="0"/> <xs:element name="Group" type="xs:string" minOccurs="0"/> <xs:element name="Comment" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<MsgDef> <Name>**b_csn1_ies**</Name> <MsgID Direction="**BOTH**" IDType="**BIN**">**00001000**</MsgID> <Comment>**CSN1 IE msg**</Comment> </MsgDef>

[5] Besides message ID the direction information is another important entry for the coder/decoder.

2.2.1.1.2 Message Items

The *Message Items* table provided by the SAPE GUI holds the MsgItem elements (sub-elements that will be contained in a message), which define the items of a message element.

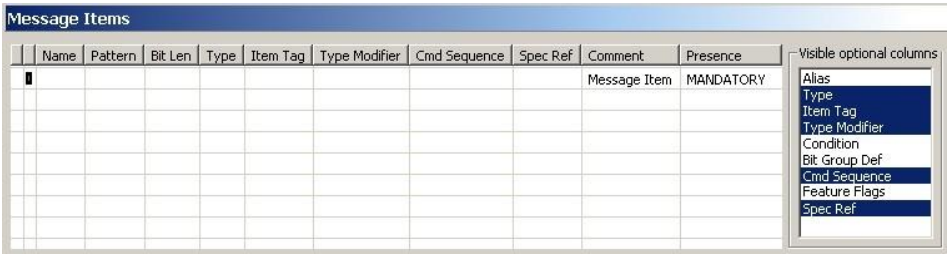


Figure 53: SAPE Message Items Table

The MsgItem element comprises all components which are needed to declare a message item. Basically a MsgItem could be the definition of a Spare element or a reference to an existing element. A complex MsgItem element could be a composition of well-assorted references and spare elements.

Each row of the *Message Items* table provided by the SAPE GUI corresponds to a message item (MsgItem). The columns of the *Message Items* table diverge a little bit from the strict conversion to support exactly one column for each child element.

The MsgItem element consists of the child elements:

- A message item is either the definition of a Spare element or a reference to an existing element. All items, except the Spare items, will be declared using an ItemLink element. It is mandatory to choice one of these alternatives.

  To reference an existing item located in the same or an external document, the ItemLink element will be used. It represents a reference to an item defined elsewhere in the same or an external document. The SAPE column labelled with the keyword *Name* offers the possibility to join a linked item by reference. The SAPE editor provides the possibility to select a new linked element from the Repository Entry and to jump to the linked element. In this case the two columns *Pattern* and *BitLen* are insignificant and therefore invalidated.

  A Spare element consists of two child elements itself: A Pattern element and the BitLen of this pattern are mandatory to define a *Spare* element. Although SAPE allows any combination of text or digits without any restrictions for both child elements the TI tool chain demands a binary coded value for the Pattern element. It is recommended to use a value facet-valid with respect to DEC for the BitLen element only. The BitLen value must match the number of bits used for the Pattern element. This condition is not proved by the XML schema validation tool and therefore the user is responsible for correct settings. In this case the columns *Name* is insignificant and therefore invalidated.

- The Alias element (optional)
  enables any combination of text or digits to identify a user specific value. This element could be used to define the new name in the case that the name of the item, under which it could be addressed within the message, should be different from the name of the linked element. By default the SAPE table doesn't provide this column, but it can be switched on easily by selecting the appropriate keyword in the *Visible Optional Columns* field.

- The Type element (optional)
  holds information about the type of an item. The content must be a valid type definition for the tool chain (e.g. C-types, CCDtypes). Part 2.2.4.2 and the corresponding file *ccd_codingtypes.h* helps to know which encoding types are supported by which version of Coder/Decoder. Any combination of text or digits is allowed. This element corresponds to a column with the same name.

Comment [K4]: Suggestion: Modification of the XML Schema so that only values facet-valid with respect to BIN for Pattern and to DEC for BitLen are possible.

- The ItemTag element (optional)
  holds the tag identifier for structured element item. This element supports locally defined values facet-valid with respect to enumeration [DEC, BIN, HEX, OCT, ALPHA].
- With the Control element (optional)
  the item could be modified (e.g. array, dependencies, conditionals). The Control element will be separated into the different kinds of control elements. Namely they are *TypeModifier, Condition, BitGroupDef* and *CmdSequence*. All of these elements are optional. The SAPE tool provides in the *Structured Element Items* table separate column for each control sub element, which can be switched to visible or turned off. The behaviour of each control mechanism is described in a separate part below.
- The optional Version element
  accepts any combination of text or digits that represents the dependency from feature flags for the definition of a specific message element. This element relates to the Feature Flags column. The entries are optional but must comply with coding rules for feature flags.
- The SpecRef element (mandatory)
  in intended to provide a reference to the part of the specification, where the item is described. Any combination of text or digits is allowed. These refer to the chapters in the GSM/GPRS/UMTS specifications.
- The Comment element (mandatory)
  enables any combination of text or digits that should make a comment. Ideally it should not be the repetition of the comment, where the element was defined; it should describe the usage of the element within the structured element. The associated XML element is mandatory and therefore a preselection is suggested.

The SAPE tool provides in the *Message Items* table an additional column labelled with the keyword *Presence* to declare the whole message item as optional, conditional or mandatory. The XML schema definition handles this presence information by the mandatory attribute Presentation, which is concatenated with each MsgItem. This attribute may take one of the values MANDATORY, OPTIONAL or CONDITIONAL. If there exists a Message Item at least one component is needed to declare this item. Therefore a preselection is suggested in the *Presence* column.

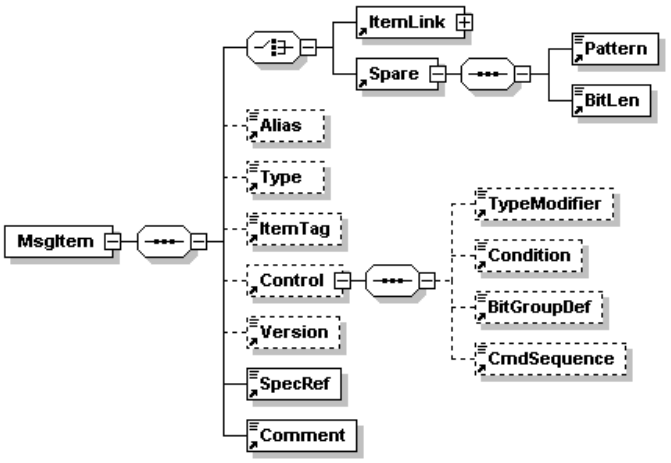
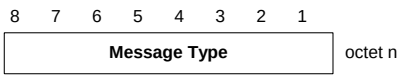
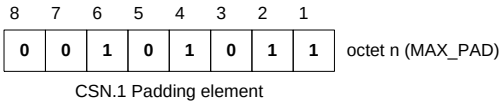


Figure 54: Model of a Message Item Element

Element MsgItem	
Children	**ItemLink Spare Alias Type ItemTag Control Version SpecRef Comment**
Used by	Element **Message**
Attributes	Name: Presentation; Type: presChoice; Use: required; Default: ; Fixed:
XML schema	<xs:element name="MsgItem"> <xs:complexType> <xs:sequence> <xs:choice> <xs:element ref="ItemLink"/> <xs:element name="Spare"> <xs:complexType> <xs:sequence> <xs:element name ="Pattern" type="xs:string"/> <xs:element name ="BitLen" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> </xs:choice> <xs:element ref ="Alias" minOccurs="0"/> <xs:element ref="Type" minOccurs="0"/> <xs:element ref="ItemTag" minOccurs="0"/> <xs:element ref="Control" minOccurs="0"/> <xs:element name="Version" type="xs:string " minOccurs="0"/> <xs:element name="SpecRef" type="xs:string"/> <xs:element name="Comment" type="xs:string"/> </xs:sequence> <xs:attribute name="Presentation" type="presChoice" use="required"/> </xs:complexType> </xs:element>
XML example *element with an ItemLink child*	<MsgItem Presentation="**MANDATORY**"> <ItemLink> **/*** Message Type */ <DocName DocType="**AIM**">**xx**</DocName> <Name>**msg_type**</Name> </ItemLink> <Type>**GSM3_V**</Type> <SpecRef /> <Comment>**Message Type**</Comment> </MsgItem>
Associated graphical presentation	 *Figure 55: Example of an Information Element (Message Type)*
XML example *element with a Spare child*	<MsgItem Presentation="**OPTIONAL**"> **/*** S_PADDING */ <Spare> <Pattern>**00101011**</Pattern> <BitLen>**8**</BitLen> </Spare> <Type>**S_PADDING**</Type> <Control> <CmdSequence>**MAX_PADD**</CmdSequence> </Control> <SpecRef>**-**</SpecRef> <Comment>**Padding**</Comment> </MsgItem>
Associated graphical presentation	 *Figure 56: Example of an Information Element (Padding)*

2.2.2 Structured Elements Section

The *Structured Elements Section* deals with the Structured Message Elements supported by the SAPE editor in table format. It handles the MsgStructElementsSection element, which comprises **all** structured elements declared within an AIM document.

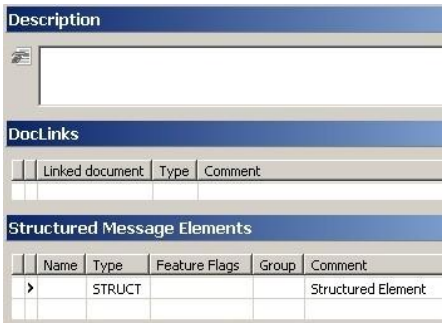


Figure 57: SAPE Structured Elements Section

This section provided by SAPE contains three different parts:

- The Description part should serve as additional information about the Structured Elements Section and should contain a textual description of the information in the Structured Message Elements table below. This part corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The third part is a table to take one to many child elements MsgStructElem (*Structured Message Elements* - mandatory) into account. They are described in a particular subsection of this document that deals with the internal structure of the structured message elements. At least one MsgStructElem element has to be present. The SAPE editor offers a table labelled *Structured Message Elements* providing a separate row for each child element MsgStructElem.

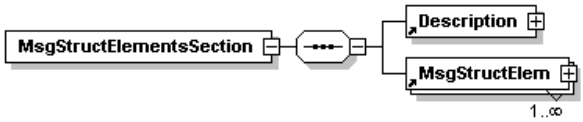


Figure 58: Model of the Structured Element Section

Element MsgStructElementsSection	
Children	Description MsgStructElem
Used by	Element AIM
XML schema	<xs:element name="MsgStructElementsSection"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="MsgStructElem" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<MsgStructElementsSection > ... MsgStructElem ... MsgStructElem ... MsgStructElem ... MsgStructE-lem</MsgStructElementsSection>

2.2.2.1 Structured Message Elements

Each row of the table labelled *Structured Message Elements* in the Structured Elements Section represents a separate child element called *MsgStructElem*. By selecting any child element respectively any row the SAPE GUI provides a set of tables needed to describe a single element sufficiently. These tables comprise all instruments being necessary to define a structured element in an AIM document. Each table relates to an XML element. Table labels and element names can be implicated easily.

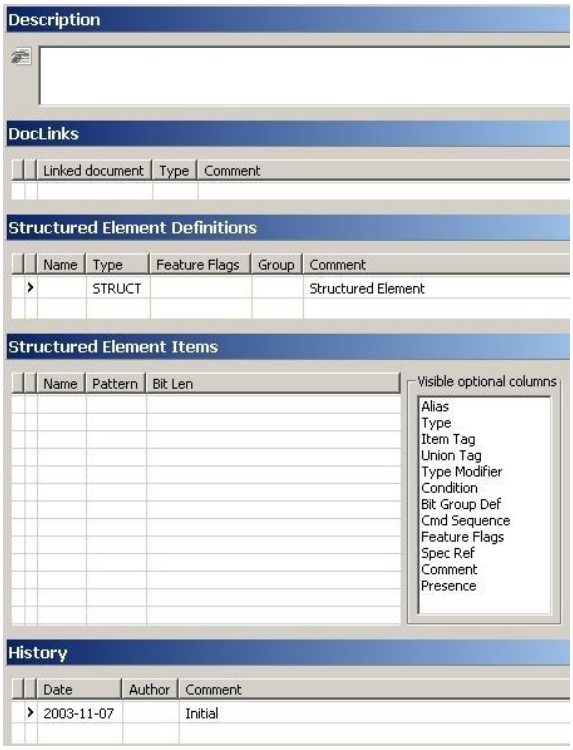


Figure 59: SAPE Structured Message Elements Tables

The MsgStructElem element serves as a sort of container itself to group a set of information defining its properties. Each row in the *Structured Message Elements* table mentioned above is associated with

another set of tables. The MsgStructElem format corresponds to other key elements in the document. Therefore also MsgStructElem elements have the mandatory elements *Description element* and *History element.*

The SAPE GUI offers three parts to take these elements into account:

- The Description part should serve as additional information about the MsgStructElem and corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The History table is intended to track changes. Each row in this table corresponds to one History element. At least one associated XML element is mandatory and therefore a preselection is suggested.

Additional tables are provided:

- The *Structured Element Definitions* table (short: MsgStructElemDef element), which is mandatory, act as a definition element. In case that the layout of the structured element should be assigned to more than only one element definition, additional MsgStructElemDef elements could be present.
- The *Structured Element Items* table supports one to many MsgStructElemItem child elements. These elements are optional. They could be attached to the MsgStructElem element if values should be assigned to the structured element.

Each of these child elements consists of nested elements itself.

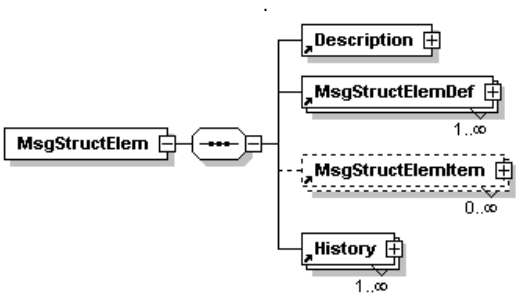


Figure 60: Model of a Structured Message Element

Element MsgStructElem	
Children	**Description MsgStructElemDef MsgStructElemItem History**
Used by	Element **MsgStructElementsSection**

XML schema	`<xs:element name="MsgStructElem">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="Description"/>`
	`<xs:element ref="MsgStructElemDef" maxOccurs="unbounded"/>`
	`<xs:element ref="MsgStructElemItem" minOccurs="0" maxOccurs="unbounded"/>`
	`<xs:element ref="History" maxOccurs="unbounded"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example (common)	`<MsgStructElem>` `... ... ... ... ... ... ... ...`

XML example ([12] part 10.5.5.15)	`<MsgStructElem>` /* Routing area identification*/
	`...`
	`<MsgStructElemDef Type="STRUCT">`
	`<Name>routing_area_identification</Name>`
	`<Comment>Routing Area Identification</Comment>`
	`</MsgStructElemDef>`
	`<MsgStructElemItem Presentation="MANDATORY">`
	`/* Mobile Country Code */`
	`</MsgStructElemItem>`
	`<MsgStructElemItem Presentation="MANDATORY">`
	`/* Mobile Network Code */`
	`</MsgStructElemItem>`
	`<MsgStructElemItem Presentation="MANDATORY">`
	`/* Location Area Code</Comment>`
	`</MsgStructElemItem>`
	`<MsgStructElemItem Presentation="MANDATORY">`
	`/* Routing Area Code</Comment>`
	`</MsgStructElemItem>`
	`<History ... History>`
	`</MsgStructElem`

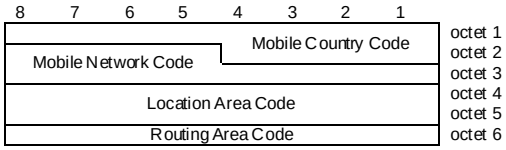
Associated GSM specification	

Table 4: Example of a Structured Message Element (Routing Area Identification Value Part)

2.2.2.1.1 Structured Element Definitions

The *Structured Element Definitions* table provided by the SAPE GUI holds the *MsgStructElemDef* element, which defines the key parameters of a simple element in an AIM document.

Figure 61: SAPE Structured Element Definitions Table

Each column in the *Structured Element Definitions* table corresponds to an XML child element of the *MsgStructElemDef* element. The column labels are equivalent to the child element names.

The MsgStructElemDef element consists of the child elements:

- The Name element serves as unique identification.
  This element is mandatory and could be used to reference this *Structured Element* by other elements.
- The optional Version element
  accepts any combination of text or digits that represents the dependency from feature flags for the definition of a specific message element. This element relates to the Feature Flags column. The entries are optional but must comply with coding rules for feature flags.
- The Group element
  could be present to declare a group where the *Structured Element* belongs. Groups can be used to force the generators of the tool chain to separate the definitions of elements into different output files. Any combination of text or digits is allowed.
- The Comment element (mandatory)
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory and therefore a preselection is suggested.

A link element is not foreseen in this format, because an element could be either defined or referenced. Therefore a link is only possible where an element will be used. As well there is no explicit length information given because it should be possible to calculate the length from the length information of the structured element items.

The SAPE tool provides in the *Structured Element Definitions* table an additional column to set the type of each *Structured Element* The XML schema definition handles this type information by the mandatory attribute *Type*. This attribute may take either the value STRUCT or the value UNION and will typically result in an item according to the C programming language.

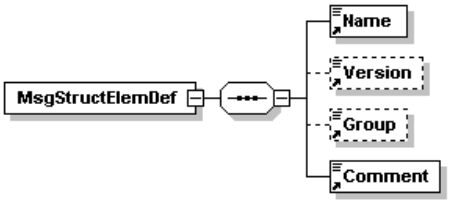


Figure 62: Model of a Structured Element Definition

Element MsgStructElemDef					
Children	**Name Version Group Comment**				
Used by	Element **MsgStructElem**				
Attributes	Name Type	Type compType-Choice	Use required	Default	Fixed
XML schema	<xs:element name="MsgStructElemDef"> <xs:complexType> <xs:sequence> <xs:element ref="Name"/> <xs:element name="Version" type="xs:string" minOccurs="0"/> <xs:element name="Group" type="xs:string" minOccurs="0"/> <xs:element name="Comment" type="xs:string"/> </xs:sequence> <xs:attribute name="Type" type="compTypeChoice" use="required"/> </xs:complexType> </xs:element>				
XML example ([12] part 10.5.5.15)	<MsgStructElemDef Type="**STRUCT**"> <Name>**routing_area_identification**</Name> <Comment>**Routing Area Identification**</Comment> </MsgStructElemDef>				

2.2.2.1.2 Structured Element Items

The *Structured Element Items* table provided by the SAPE GUI holds the *MsgStructElemItem* elements, which define the items of a structured or combined element. The MsgStructElemItem element comprises all components which are needed to declare a structured message element item. Basically a MsgStructElemItem could be the definition of a Spare element or a reference to an existing element. A complex MsgStructElemItem element could be a composition of well-assorted references and spare elements.

Each row of the *Structured Element Items* table provided by the SAPE GUI corresponds to a single structured element item (MsgStructElemItem). The columns of the *Structured Element Items* table diverge a little bit from the strict conversion to support exactly one column for each child element.

Figure 63: SAPE Structured Element Items Table

The MsgStructElemItem element consists of the child elements:

- A structured message element item is either the definition of a Spare element or a reference to an existing element. All items, except the Spare items, will be declared using an ItemLink element. It is mandatory to choice one of these alternatives.

  To reference an existing item located in the same or an external document, the ItemLink element will be used. It represents a reference to an item defined elsewhere in the same or an external document. The SAPE column labelled with the keyword *Name* offers the possibility to join a linked item by reference. The SAPE editor provides the possibility to select a new linked element from the Repository Entry and to jump to the linked element. In this case the two columns *Pattern* and *BitLen* are insignificant and therefore invalidated.

  A Spare element consists of two child elements itself: A Pattern element and the BitLen of this pattern are mandatory to define a *Spare* element. Both child elements allow any combination of text or digits without any restrictions. But it is recommended to use values facet-valid with respect to enumeration [DEC, BIN, HEX, OCT] only. In this case the columns *Name* is insignificant and therefore invalidated.

  Comment [K5]: Suggestion: Modification of the XML Schema so that only values facet-valid with respect to enumeration [DEC, BIN, HEX, OCT] are possible.

- The Alias element (optional)
  enables any combination of text or digits to identify a user specific value. This element could be used to define the new name in the case that the name of the item, under which it could be addressed within the message, should be different from the name of the linked element. By default the SAPE table doesn't provide this column, but it can be switched on easily by selecting the appropriate keyword in the *Visible Optional Columns* field.
- The Type element (optional)
  holds information about the type of an item. The content must be a valid type definition for the tool chain (e.g. C-types, CCDtypes). Any combination of text or digits is allowed.
- The ItemTag element (optional)
  holds the tag identifier for structured element item. This element supports locally defined values facet-valid with respect to enumeration [DEC, BIN, HEX, OCT, ALPHA].

- The UnionTag element (optional)
  is a tag identifier of union elements that indicates which union element of all possible elements is present.
- With the Control element (optional)
  the item could be modified (e.g. array, dependencies, conditionals). The Control element will be separated into the different kinds of control elements. Namely they are *TypeModifier, Condition, BitGroupDef* and *CmdSequence*. All of these elements are optional. The SAPE tool provides in the *Structured Element Items* table a separate column for each control sub element, which can be switched to visible or turned off. The behaviour of each control mechanism is described in a separate part below.
- The optional Version element
  accepts any combination of text or digits that represents the dependency from feature flags for the definition of a specific message element. This element relates to the Feature Flags column. The entries are optional but must comply with coding rules for feature flags.
- The SpecRef element (mandatory)
  in intended to provide a reference to the part of the specification, where the item is described. Any combination of text or digits is allowed. These links refer to the chapters in the GSM/GPRS/UMTS specifications.
- The Comment element (mandatory)
  enables any combination of text or digits that should make a comment. Ideally it should not be the repetition of the comment, where the element was defined; it should describe the usage of the element within the structured element. The associated XML element is mandatory and therefore a preselection is suggested.

The SAPE tool provides in the *Structured Element Items* table an additional column labelled with the keyword *Presence* to declare the whole structured element item as optional, conditional or mandatory. The XML schema definition handles this presence information by the mandatory attribute Presentation, which is concatenated with each *MsgStructElemItem*. This attribute may take one of the values MANDATORY, OPTIONAL or CONDITIONAL. If there exists a *Structured Element Item* at least one component is needed to declare this item. Therefore a preselection is suggested in the *Presence* column.

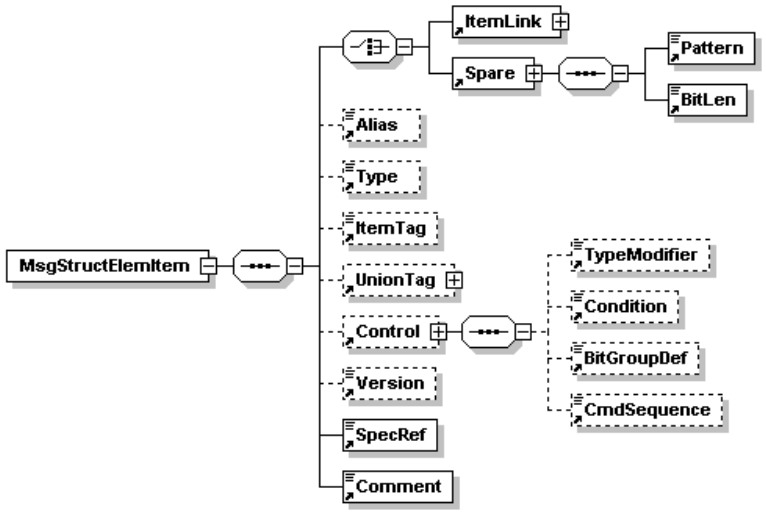


Figure 64: Model of a Structured Element Item

Element MsgStructElemItem	
Children	**ItemLink Spare Alias Type ItemTag UnionTag Control Version SpecRef Comment**
Used by	Element **MsgStructElem**
Attributes	Name: Presentation; Type: presChoice; Use: required; Default: ; Fixed:
XML schema	<xs:element name="MsgStructElemItem"> <xs:complexType> <xs:sequence> <xs:choice> <xs:element ref="ItemLink"/> <xs:element name="Spare"> <xs:complexType> <xs:sequence> <xs:element name ="Pattern" type="xs:string"/> <xs:element name ="BitLen" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> </xs:choice> <xs:element ref="Alias" minOccurs="0"/> <xs:element ref="Type" minOccurs="0"/> <xs:element ref="ItemTag" minOccurs="0"/> <xs:element ref="UnionTag" minOccurs="0"/> <xs:element ref="Control" minOccurs="0"/> <xs:element name="Version" type="xs:string " minOccurs="0"/> <xs:element name="SpecRef" type="xs:string"/> <xs:element name="Comment" type="xs:string"/> </xs:sequence> <xs:attribute name="Presentation" type="presChoice" use="required"/> </xs:complexType> </xs:element>
XML example ([12] part 10.5.5.15)	<MsgStructElemItem Presentation="**MANDATORY**"> <ItemLink> <DocName DocType="**AIM**">**gmm**</DocName> <Name>**mcc**</Name> </ItemLink> <Type>**BCD_NOFILL**</Type> <Control> <TypeModifier>**[3]**</TypeModifier> </Control> <SpecRef>**-**</SpecRef> <Comment>**Mobile Country Code**</Comment> </MsgStructElemItem>
Associated graphical presentation	8 7 6 5 4 3 2 1 MCC digit 2 | MCC digit 1 — octet 1 MCC digit 3 — octet 2

Table 5: Example of a Structured Message Element Item (Mobile Country Code)

2.2.3 Basic Elements Section

The *Basic Element Section* deals with the *Basic Message Elements* table provided by the SAPE editor. It handles the *MsgBasicElementsSection* element, which comprises **all** basic elements declared within an AIM document.

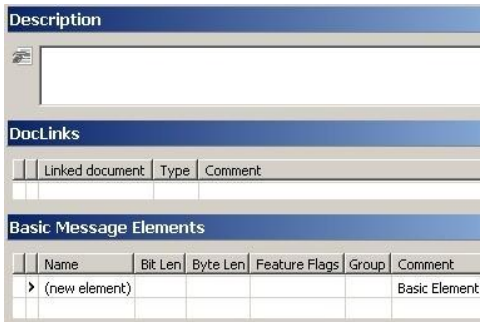


Figure 65: SAPE Basic Elements Section

The *MsgBasicElementsSection* serves as a container for all basic elements. For these purpose the SPAE GUI provides three different parts:

- The Description part should serve as additional information about the Basic Elements Section and should contain a textual description of the information in the Basic Message Elements table below. This part corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The third part is a table to take one to many child elements MsgBasicElem (*Basic Message Element* - mandatory) into account. They are described in a particular subsection of this document, which deals with the internal structure of the *Basic Element Definitions*. At least one MsgStructElem element has to be present. The SAPE editor offers a table labelled *Basic Message Elements* providing a separate row for each child element MsgBasicElem.

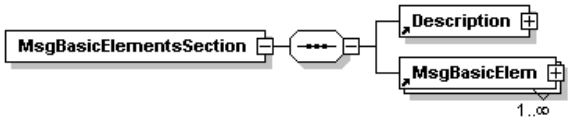


Figure 66: Model of the Basic Elements Section

Element MsgBasicElementsSection	
Children	**Description** **MsgBasicElem**
Used by	Element **AIM**
Source	<xs:element name="MsgBasicElementsSection"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="MsgBasicElem" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<MsgBasicElementsSection> ... MsgBasicElem ... MsgBasicElem ... MsgBasicElem ... MsgBasicE-lem</MsgBasicElementsSection>

2.2.3.1 Basic Message Elements

Each row of the table labelled *Basic Message Elements* in the Basic Elements Section represents a separate child element called *MsgBasicElem* . By selecting any child element respectively any row the SAPE GUI provides a set of tables needed to describe a single element sufficiently. These tables comprise all instruments being necessary to define a structured element in an AIM document. Each table relates to an XML element. Table labels and element names can be implicated easily.

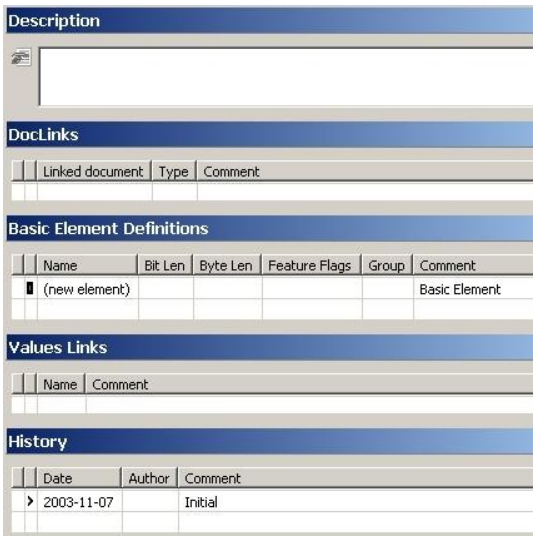


Figure 67: SAPE Basic Message Elements Tables

The MsgBasicElem element serves as a sort of container itself to group a set of information defining its properties. Each row in the *Basic Message Elements* table mentioned above is associated with another set of tables. The MsgBasicElem format corresponds to other key elements in the document. Therefore also MsgBasicElem elements have the mandatory elements *Description element* and *History element*.

The SAPE GUI offers three parts to take these elements into account:

- The Description part should serve as additional information about the MsgBasicElem and corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The History table is intended to track changes. Each row in this table corresponds to one History element. At least one associated XML element is mandatory and therefore a preselection is suggested.

Additional tables are provided:

- The Basic Element Definitions table (short: MsgBasicElemDef element), which is mandatory, act as a definition element.
- The ValuesLink table supports one to many *Values Link* child elements. These elements are optional. They could be attached to the MsgBasicElem element if values should be assigned to the basic element. This element is optional.

Each of these child elements consists of nested elements itself.

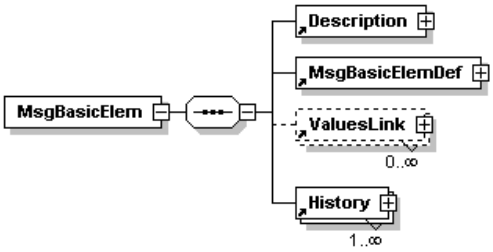


Figure 68: Model of a Basic Message Element

Element MsgBasicElem	
Children	Description MsgBasicElemDef ValuesLink History
Used by	Element MsgBasicElementsSection
XML schema	<xs:element name="MsgBasicElem"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="MsgBasicElemDef"/> <xs:element ref="ValuesLink" minOccurs="0" maxOccurs="unbounded"/> <xs:element ref="History" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example (common)	< MsgBasicElem > ... MsgBasicElemDef ... MsgBasicElemDef ValuesLink ... ValuesLink ... ValuesLink ... ValuesLink ... ... MsgBasicElem
XML example ([12] part 10.5.5.15)	<MsgBasicElem> /* Mobile Country Code Digit*/ <Description ... Description> <MsgBasicElemDef> <Name>**mcc**</Name> <BitLen>**4**</BitLen> <Comment>**Mobile Country Code**</Comment> </MsgBasicElemDef> ... </MsgBasicElem>

2.2.3.1.1 Basic Element Definitions

The *Basic Element Definitions* table provided by the SAPE GUI holds the *MsgBasicElemDef* element, which defines the key parameters of a simple element in an AIM document.

Each column in the *Basic Element Definitions* table corresponds to an XML child element of the *MsgBasicElemDef* element. The column labels are equivalent to the child element names

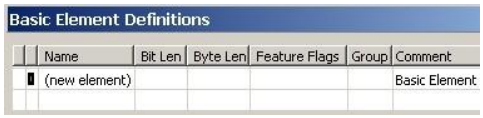


Figure 69: SAPE Basic Element Definitions Table

The *MsgBasicElemDef* element consists of the following child elements:

- The Name element serves as unique identification.
  This element is mandatory and could be used to reference this *Basic Element* by other elements.
- The length of the *Basic Element* is mandatory and could be given either in unit of bits or in units of bytes. Therefore one of the two elements BitLen or ByteLen must be present. The XML

schema description allows any combination of text or digits without any restrictions. But it is recommended to use values facet-valid with respect to enumeration [DEC, BIN, HEX, OCT] only.

Comment [K6]: Suggestion: Modification of the XML Schema so that only values facet-valid with respect to enumeration [DEC, BIN, HEX, OCT] are possible.

- The optional Version element
  accepts any combination of text or digits that represents the dependency from feature flags for the definition of a specific message element. This element relates to the Feature Flags column. The entries are optional but must comply with coding rules for feature flags.
- The Group element
  could be present to declare a group where the *Basic Element* belongs. Groups can be used to force the generators of the tool chain to separate the definitions of elements into different output files. Any combinations of text or digits are allowed.
- The Comment element (mandatory)
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory and therefore a preselection is suggested.

A link element is not foreseen in this format, because an element could be either defined or referenced. Therefore a link is only possible where an element will be used.

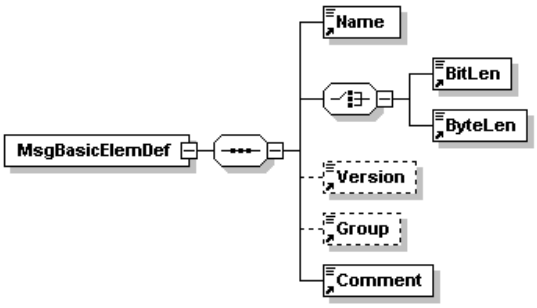


Figure 70: Model of a Basic Element Definition

Element MsgBasicElemDef	
Children	**Name BitLen ByteLen Version Group Comment**
Used by	Element **MsgBasicElem**

XML schema	`<xs:element name="MsgBasicElemDef">`
	`  <xs:complexType>`
	`  <xs:sequence>`
	`  <xs:element ref="Name"/>`
	`  <xs:choice>`
	`  <xs:element name="BitLen" type="xs:string"/>`
	`  <xs:element name="ByteLen" type="xs:string"/>`
	`  </xs:choice>`
	`  <xs:element name="Version" type="xs:string" minOccurs="0"/>`
	`  <xs:element name="Group" type="xs:string" minOccurs="0"/>`
	`  <xs:element name="Comment" type="xs:string"/>`
	`  </xs:sequence>`
	`  </xs:complexType>`
	`</xs:element>`

XML example ([12] part 10.5.5.15)	`<MsgBasicElemDef>` /* Mobile Country Code Digit*/
	`  <Name>`**mcc**`</Name>`
	`  <BitLen>`**4**`</BitLen>`
	`  <Comment>`**Mobile Country Code**`</Comment>`
	`</MsgBasicElemDef>`

2.2.4 Nontrivial AIM Specific Sub-Elements

Some AIM specific sub-elements, which may occur in different context, require more detailed explanation. These elements are listed here to provide additional information about the data they may contain. This section should serve primarily as a reference for these elements.

2.2.4.1 Control

The Control element is the most complex of all. It contains instructions used by the TI tool chain how to define or interpret the associated object in question.

This optional element is intended to hold any kind of control information to classify an item. With the information contained in the Control element an item could be modified in order to achieve different constructions, such as arrays, pointers and optional parameters etc. The SAPE tool provides in the *Structured Element Items* table as far as in the *Message Items* table a separate column for each control sub element, which can be switched to visible or turned off.

Depending on the kind of control information classification is done. According to the different types of control information a specific sub elements is used. Namely they are *TypeModifier, Condition, BitGroupDef* and *CmdSequence*. These child elements take alphanumerical data and are optional, too

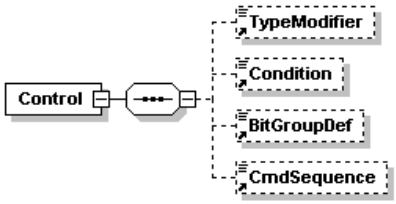


Figure 71: Model of a Control Element

Element Control	
Children	**TypeModifier Condition BitGroupDef CmdSequence**
Used by	Elements **MsgItem MsgStructElemItem**

XML schema	`<xs:element name="Control">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element name="TypeModifier" type="xs:string" minOccurs="0"/>`
	`<xs:element name="Condition" type="xs:string" minOccurs="0"/>`
	`<xs:element name="BitGroupDef" type="xs:string" minOccurs="0"/>`
	`<xs:element name ="CmdSequence" type="xs:string" minOccurs="0"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<Control>`
	`<TypeModifier>`**[2..3]**`</TypeModifier>`
	`</Control>`

2.2.4.1.1 Type Modifier Element

The *TypeModifier* element associated with a corresponding SAPE column enables constructions of element arrays. Arrays can only be found in declarations of messages or of structured message elements of content type **STRUCT**.

Note: Arrays of unions are not supported.

Each item in a table being an array needs its own *TypeModifier* to define the number of elements in the array. All elements of one array have the same type given by the item corresponding to the Type column respectively to the content type specified for the element.

Syntax Definition:

Comment [K7]: Unions must be encapsulated in a structure in order to create an array. The structure requirement is due to the extra union controller ("ctrl_") element inserted by the TI tool chain; this element is outside the union, and thus needs a structure to contain it. ->Additional investigation needed: SAPE handles this problem as if unions are structures. But how does the TI toll chain work?

TypeModifier ::= [DYN | PTR] "[" MinimumElementNumber
[".." MaximumElementNumber] "]" | "0.." MaximumElementNumber "]"

MinimumElementNumber ::= [.] Constant | BasisMessageElement [ArithmOp Number] | TakeCmdSeq

MaximumElementNumber ::= Constant | BasisMessageElement [ArithmOp Number] | TakeCmdSeq

Constant ::= Number | ConstantAlias

Number ::= (NonZeroNum {Num})

NonZeroNum ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Num ::= **0** | NonZeroNum

ConstantAlias ::= /* Reference to an Alias element of the Constants part */

BasisMessageElement ::= /* Reference to a MsgStructElemItem in case of array declaration in a *structured element items* table or
Reference to a child of another MsgItem in case of array declaration in a *message items* table
Note: Only References to items of integral type are allowed. */

TakeCmdSeq ::= "(" (TakeCmd | LengthTakeCmd) ")" [ArithmOp Number]

TakeCmd[6] ::= **TAKE** ","RegNum
/* Reference to a value held in a register */

RegNum ::= [**1**] Num [7]

ArithmOp ::= **+** | **-** | ***** | **/**

Different kinds of arrays can be distinguished: An array may have a fixed number of elements, a variable number of elements or a number of elements depending on other items. Besides it is possible to specify bit arrays.

Arrays with a fixed number of elements need the *MinimunElementNumber* only. This value can be a positive number of integral types or the Alias of global constants with a value larger than zero.

Structured Element Items / Message Items			
Name	**Type**	**Type Modifier**	**Comment**
s1_fix_arr	CSN1_S1	[3]	CSN1_S1 fixed array with 3 elements

Table 6: Example of an Array with Fixed Length

Arrays with a variable number of elements need an upper limit named *MaximumElementNumber* and a lower limit, which is either zero or a *MinimumElementNumber* and must be less than the *Maximum-ElementNumber*. Each limit can be a positive number of integral types or the Alias of a global constant.

Structured Element Items / Message Items			
Name	**Type**	**Type Modifier**	**Comment**
s1_var_arr	CSN1_S1	[0..3]	CSN1_S1 variable array (0 up to 3 elements)

Table 7: Example of an Array with Variable Length

[6] TakeCmdSeq specification: usage according to part *Command Sequence Element*.
[7] The maximum register number depends on the internal constant MAX_UPN_STACK_SIZE (stack size for UPN calculator) defined in ccd_globs.h; current value: MAX_UPN_STACK_SIZE = 20.

If the number of elements depends on other items the appropriate limit needs to be replaced by a reference. This reference may refer either to an MsgStructElemItemin case of an array declaration in a *structured element items* table or to a child of another MsgItem in case of array declaration in a *message items* table.

Structured Element Items / Message Items			
Name	**Type**	**Type Modifier**	**Comment**
s1_c	CSN1_S1		CSN1_S1 repetition counter
s1_calc_arr	CSN1_S1	[s1_c..5]	CSN1_S1 calculated array (s1_c up to 5 elements)

Table 8: Example of an Array with Variable Length Depending on Another Item

To specify a **bit array** the beginning of the *MinimunElementNumber* is marked with a single dot. A bit array may have either a fixed number or a variable number of elements. The rules to define the amount of elements comply with conditions mentioned above. **Note, that the length of a bit array is determined by limits giving in bits**. Make sure that the type given by the item corresponding to the Type column respectively to the content type specified for the element allows an amount of bits specified by the upper range value *MaximumElementNumber*.

Structured Element Items / Message Items			
Name	**Type**	**Type Modifier**	**Comment**
gsm5_v	GSM5_V	[.0..24]	GSM5_V IE; bit array (length: 0 up to 24 bits)

Table 9: Example of an Array with Variable Length Depending on Another Item

Dynamic Arrays or Pointer Types

The control keywords **DYN** or **PTR** allow the specification of dynamic size arrays. The amount of memory is dynamically allocated depending on the number of elements. Each keyword implicates a certain C code generation. The dynamic array specifier **DYN** is code transparent, i.e. in the type definition the element name is used without any prefix but a valid flag precedes this element. If an element is associated with the keyword **PTR** the dynamic size array identifier is non-code transparent, i.e. in the type definition the element name is concatenated with the prefix **ptr**_ but no valid flag precedes this element. In case of missing this optional element the value of the pointer is set to NULL. (cf. part 2.3.5.2.3).

Conclusion

The C expressions generated by the TI tool chain reveals the difference between the mentioned types of arrays.

- Arrays with a **fixed** number of elements:
  In the element description the Type Modifier Element consists of a minimum element number only. The C expression is given by the example below. `short_name` corresponds to the Name column; `T_SHORT_NAME` is replaced by an item corresponding to the Type column respectively to the content type specified for the element. The amount of memory is statically allocated depending on `MINIMUM_ELEMENT_NUMBER`

  ```
  T_SHORT_NAME short_name[MINIMUM_ELEMENT_NUMBER];
  ```

- Arrays with a **variable** number of elements:
  In the element description a minimum and a maximum number of elements is used in the Type Modifier Element. The C expression is given by the example below. `short_name` corresponds to the Name column; `T_SHORT_NAME` is replaced by an item corresponding to the Type column respectively to the content type specified for the element. The amount of memory is statically allocated depending on `MAXIMUM_ELEMENT_NUMBER`. The type of `c_short_name` depends on the value of `MAXIMUM_ELEMENT_NUMBER`. The parameter `c_short_name` contains information about

the number of elements currently present in the variable size array. Therefore only the array elements [0; c_short_name - 1] contain valid data.

```
U8 c_short_name;
T_SHORT_NAME short_name[MAXIMUM_ELEMENT_NUMBER];
```

- Arrays with a **dynamic** size:
  The number of elements is variable but the amount of memory for the C expression is dynamically allocated depending on the number of elements currently used. The Type Modifier Element should be one the keywords **DYN** or **PTR** followed by a range definition.

  The keywords **DYN** and **PTR** affect the generated C expressions shown by the examples below. `short_name` corresponds to the Name column; `T_SHORT_NAME` is replaced by an item corresponding to the Type column respectively to the content type specified for the element. The type of `c_short_name` depends on the upper range value (`MAXIMUM_ELEMENT_NUMBER`). The parameter `c_short_name` contains information about the number of elements currently present in the dynamic size array.

  If the keyword **DYN** is used:

  ```
  U8 c_short_name;
  T_SHORT_NAME * short_name;
  ```

  If the keyword **PTR** is used:

  ```
  U8 c_short_name;
  T_SHORT_NAME * ptr_short_name;
  ```

  In both cases the result is the generation of a pointer of the specified type, only the naming of the pointer is different. The difference in the use of the elements between **DYN** and **PTR** declarations lies more in the behaviour when elements are declared optional (cf. part 6

Generated C-Code Header Files). When the keyword **PTR** is used for an optional element no valid flag is added to the generated C expressions. Instead the element is not present if the value of the pointer is NULL.

Please note that it is not possible to make array of pointers.

2.2.4.1.2 Condition Element

The *Condition* element is one child of the control element. It is associated with a column of the same name provided by SAPE. This element helps to specify the conditions whether an item shall be included in a message or not.

The inclusion of the item by the sender depends on conditions specified in the relevant protocol specification. The receiver decides to expect by means of conditions that the item is present or absent. These conditions depend only on the content of the message itself. Therefore it is necessary to provide sufficient design features. The condition is checked at runtime.

Syntax Definition:

Condition	::=	Expression { LogOp Expression }
Expression[8]	::=	DataObjRef RelOp Value
DataObjRef	::=	/* Reference to a MsgStructElemItem in case of condition declaration in a *structured element items* table or Reference to a child of another MsgItem in case of condition declaration in a *message items* table */
Value	::=	DataObjRef | Number | ConstantAlias |
Number	::=	(NonZeroNum {Num}) | 0
NonZeroNum	::=	**1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
Num	::=	**0** | NonZeroNum
RelOp	::=	"=" | "#" | "<" | ">"
LogOp	::=	**AND** | **OR** | **XOR**
ConstantAlias	::=	/* Reference to an Alias element of the Constants part */
ArithmOp	::=	**+** | **-** | ***** | **/**

Assigning a Boolean expression to the associated *Condition* element qualifies an item. This qualification shall be taken to mean that the item is present only if the qualifier evaluates to TRUE. **Note: The logical expression which is situated rightmost will be processed first. Processing of additional logical expressions will continue from right to left. It is not possible to**

An example of this application is the information element *PBCCH Description* (RR protocol [14.]). The following table shows how to specify this part:

[8] The table below lists the operators and their precedence and associativity values. The highest precedence level is at the top of the table.

Symbol	Name or Meaning	Associa-tivity
*	Multiply	Left to right
/	Divide	
+	Add	Left to right
-	Subtract	
= # < >	Equal, Not Equal, Less than, Greater than	

Structured Element Items			
Name	**Type**	**Condition**	**Comment**
pb			Pb
tsc			Training Sequence Code
tn			Time Slot Number
flag			Flag
flag2		{flag=0}	Flag2 – only present if Flag is set to FALSE
arfcn		{flag=0 AND flag2=1}	ARFCN - only present if Flag is set to FALSE and flag2 is set to TRUE
maio		{flag=1}	Mobile Allocation Index Offset

Table 10: SAPE Table Belonging to RR PBCCH Description *information element*

2.2.4.1.3 Command Sequence Element

The *CmdSequence* element is another child of the Control element relating to Command Sequence expressions. SAPE provides a column with the same name as this element. The *CmdSequence* element allows starting specific predefined actions on of MsgStructElemItem or MsgItem elements. It tells the TI tool chain how to deal with an item.

Syntax Definition:

CmdSequence ::= CmdSeq | PadCmdSeq

CmdSeq ::= PosCmdSeq | RegCmdSeq | StackManipulation { "," (PosCmdSeq | RegCmdSeq | StackManipulation) }

PosCmdSeq ::= **GETPOS** | **SETPOS**

RegCmdSeq ::= "(" KeepCmdSeq | TakeCmdSeq | MaxCmdSeq ")"

KeepCmdSeq ::= **KEEP** "," RegNum

TakeCmdSeq ::= "(" (TakeCmd | LengthTakeCmd) ")" [ArithmOp Number]

TakeCmd ::= **TAKE** ","RegNum
/* Reference to a value held in a register */

LengthTakeCmd ::= **LTAKE**

MaxCmdSeq ::= **MAX** ","RegNum

PadCmdSeq ::= Number | ConstantAlias

Number ::= (NonZeroNum {Num}) | 0

NonZeroNum ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Num ::= **0** | NonZeroNum

ConstantAlias ::= /* Reference to an Alias element of the Constants part */

RegNum[Error! Bookmark not defined.] ::= [**1**] Num

StackManipulation ::= **:** | **^** | **+** | Number

This child elements associated with a corresponding SAPE column support a set of specific mathematical operations. These operations are performed by a RPN calculator built into TI's runtime tools. The RPN calculator uses a Data Stack (not unlike the venerable HP48) that leads to a notation in which operands precede operators. This is a postfix notation often called RPN or *Reverse Polish Notation*[9].

This stack machine offers the support of quite complex expressions: It provides a set of commands to perform arithmetic and logical functions. The functions are intended to integer arithmetic. Numbers or addresses are placed onto the stack (32 bit processing) and the control elements contain operators which act on them to produce a desired result. In general, the operands are removed (popped) from the stack and the results are left on the stack. When numbers are pushed onto or popped off the stack, the remaining numbers are not moved. Instead, a pointer is adjusted to indicate the next unused cell in a static memory array.

[9] This notation is based on the "Sentential Calculus" as developed by Professor Jan Lukasiewicz in the 1920s whilst working at Warsaw University *(Lukasiewicz, 1963)*.

For example the calculator allows movements of the bit-pointer in a message and comparisons. It is recommended to use these stack pointer operations very carefully. ***Knowledge about the current position of the stack pointer must go without saying as far as a correct initial value.***

The **KeepCmdSeq** expression enables to store a value of a variable in a certain CCD register. This feature is relevant if an item value shall outstay the lifetime of an MsgStructElemItem or MsgItem of which the item belongs to. Only registers numbered from 0 to MAX_UPN_STACK_SIZE are allowed. The length information of a TLV element will be written to Register no. 0. Therefore this register is advised against other usage.

The **TakeCmdSeq** expression enables CCD to get a value of the specified CCD register.

The **MaxCmdSeq** statement provides a comparison between a variable and a value stored in a certain CCD register. Then the maximum will be kept in the selected register.

The **PadCmdSeq** expression determines the maximum length of spare padding bits. This control element is always associated with a definition of a Spare element. The used number **n** means if a Structured Message Element or a Message consists of less than **n** bytes the remaining part shall be filled up with the bit pattern given by the Pattern element. The usage of number **n = 0** denotes a special case: If the message doesn't end on octet boundary the remaining part up to the next octet boundary shall be filled up with the rest of the bit pattern given by the Pattern element on the appropriate bit positions. The following example should point up this feature.

Structured Element Items						
Name	**Pattern**	**Bit Len**	**Type**	**Type Modifier**	**CmdSequence**	**Comment**
freq_range			CSN1_S1	[0..MAX_RANGE]		Range Limits
arfcn			CSN1_S1	[0..MAX_ARFCN]		BA Frequency
	00101011	8	S_PADDING		0	Spare Padding

Table 11: SAPE Table Belonging to BA List Pref information element (RR protocol [14.])

Assumption: The last bits of item *arfcn* ends on octet **m** bit position 6**,** then bit 5 down to 1 will be filled with s_padding bits:

	8	7	6	5	4	3	2	1	
octet m	x	x	x	.	.	.	.	.	Rest of arfcn[ ]
	.	.	.	0	1	0	1	1	Padding Bits

The processing order of items within a single octet often differs from the order of bit transmission. The **PosCmdSeq** advices the TI tool chain where to find (in case of decoding) or where to place (in case of encoding) an item in a bit string.

A message is a bit string which is very often described as a succession of octets. Octets in a message or in a part are numbered from 1 onward, starting at the beginning of the bit string. Bits in octets are numbered from 8 down to 1, starting at the beginning of the octet.

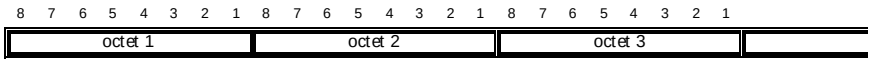


Table 12: Order of Bit Transmission

The table representations orders message octets from the top of a table downwards. Bits in octets are presented with the first bit on the left side.

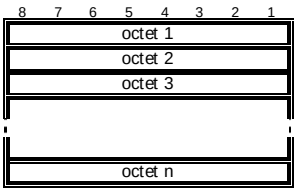


Table 13: Order of Message Octet – Table Representation

The **PosCmdSeq** enables to push and to pop information of bit stream pointer positions to the stack. So it is possible to have an impact on certain stack data and therefore on the element where the bi stream pointer refers to.

The following table gives a survey of the Control parameters to specify the calculation steps for the RPN calculator and describes their operation. Rather the logical operations (AND, OR and XOR) belong to part 2.2.4.1.2 , but they are mentioned here for the sake of completeness.

Control parameters	**Description of operation**	**Example**
<Number>	• Pushes the constant value <Number> onto the current stack position (stack position **n**) and • Increases the stack pointer (new stack position **n+1**) (cf. PUSH operation)	(GETPOS,:,**6**,+,SETPOS)
<DataObjRef>	• Pushes the content of the C-structure variable <DataObjRef> onto the stack starting with the current stack position (stack position **n**) and • Moves the stack pointer to the next available stack position (cf. PUSH operation)	
SETPOS	• Current stack position: **n** • Reads the element from the last used stack position (stack position **n-1**), • **Sets** the position of the bit stream pointer to this value and • Decreases the stack pointer (new stack position **n-1**) (cf. POP operation)	(GETPOS,:,6,+,**SETPOS**)
GETPOS	• Pushes the position of the bit stream pointer onto the current stack position (stack position **n**) and • Increases the stack pointer (new stack position **n+1**) (cf. PUSH operation)	(**GETPOS**,:,6,+,SETPOS)
:	• Current stack position: **n** • **Duplicates** the element of the last used stack position (stack position **n-1**) • Pushes it onto the current stack position and • Increases the stack pointer (new stack position **n+1**) (cf. PUSH operation)	(GETPOS,**:**,6,+,SETPOS)
^	• Current stack position: **n** • **Swaps** the two elements of the of the last used stack positions (stack positions **n-1** and **n-2**) • New stack position: **n**	

+ - * /	• Current stack position: **n**
	• Reads the last two elements from the last used stack positions (stack positions **n-1** and **n-2**)
	• Executes the appropriate arithmetic operation and pushes the result onto the stack (stack position **n-2**):
	elem(n-1) + elem(n-2);
	elem(n-1) - elem(n-2);
	elem(n-1) * elem(n-2);
		(GETPOS,:,6,**+**,SETPOS)

	elem(n-1) / elem(n-2) • Performs CCD error handling in cases of division by zero; • Sets the stack pointer to stack position **n-1** • **Note: Division by 0 caused a CCD Error and stops the encoding/decoding process!**	
& |	• Current stack position: **n** • Reads the last two elements from the last used stack positions (stack positions **n-1** and **n-2**) • Performs the appropriate bit operation (either binary AND or binary OR) and pushes the result onto the stack (stack position **n-2**) • Sets the stack pointer to stack position **n-1**	
AND OR XOR	• Current stack position: **n** • Reads the last two elements from the last used stack positions (stack positions **n-1** and **n-2**) • Performs the appropriate logical operations: AND, OR and XOR and pushes the result (1 in case of TRUE or 0 in case of FALSE) onto the stack (stack position **n-1**) • Sets the stack pointer to stack position **n-1**	{flag=1 **AND** flag2=1 **OR** flag=0}

= # < >	• Current stack position: **n**
	• Reads the last two elements from the last used stack positions (stack positions **n-1** and **n-2**):
	elem(n-1) = elem(n-2);
	elem(n-1) # elem(n-2);
	elem(n-1) < elem(n-2);
	elem(n-1) > elem(n-2)
	• Executes numerical comparison (# used for different) and pushes the result (either TRUE or FALSE) onto the stack (stack position **n-2**)
	• Sets the stack pointer to stack position **n-1**
		(KEEP,1) {n_r_cells # 0}

KEEP	• Current stack position: **n** • Reads the element from the last used stack position (stack position **n-1**) • Copies the value in the KEEP register and • Sets the stack pointer to stack position **n-1** (Opposite of TAKE; cf. POP operation)	**KEEP,1**
LTAKE	• Current stack position: **n** • Copies the L part of a TLV element from the KEEP register, converts the value to multiple of bits, pushes the result onto the current stack position, and • Increases the stack pointer (new stack position **n+1**) (Specific case of TAKE, cf. PUSH operation)	[(**LTAKE**/12)..MAX_N_PDU_NUMBER_LIST]
TAKE	• Current stack position: **n** • Takes a value from the KEEP register and pushes the value onto the current stack position • Increases the stack pointer (new stack position **n+1**) (Opposite of KEEP, cf. PUSH operation)	[.(**TAKE,1**)+1..8]
MAX	• Current stack position: **n** • Reads the element from the last used stack position (stack position **n-1**) and compares this value with the one stored in the KEEP register. • Pushes the higher value to the KEEP register. • Sets the stack pointer to stack position **n-1** (cf. POP operation)	**MAX,2**
22	• Determines the maximum length of spare padding bits	S_PADDING .00101011 (**22**)

Table 14: Control parameters to specify the calculation steps for the RPN calculator

Examples:

1) *SMS - RP-User data*

The RP-User data field contains the TPDU and is mandatory in a RP-DATA message. The element has a variable length, up to 232 octets. The TP-Message-Type-Indicator is a 2-bit field, located within bits no 0 and 1 of the first octet of all PDUs.

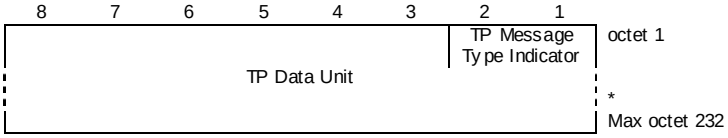


Table 15: RP-User data *IE value part*

The following table shows how to specify this part:

Structured Element Items					
Name	**Pattern**	**Bit Len**	**Type**	**CmdSequence**	**Comment**
tp_mti				GETPOS,:,6,+,SETPOS	TP Message Type Indicator
tpdu			GSM5_V	SETPOS	TP Data Unit

Table 16: SAPE Table Belonging to the RP-User data *IE value part*

Explanation:

TP Message Type Indicator	GETPOS	Determines the current bit stream pointer position (bpp: m) and writes this address information to the RPN stack (rpn: n)
	:	Duplicates the address information (rpn: n+1)
	6	Is pushed to the next stack position (rpn: n+2)
	+	Adds the constant value 6 to the bit stream pointer position and saves the result (bpp: m+6) instead of the duplicate of the old bit stream pointer position on the stack (rpn: n+1)
	SETPOS	Reads the last element from the stack (rpn: n+1), sets the position of the bit stream pointer to this value (bpp: m+6) and sets the stack pointer to stack position (rpn: n+1)
TP Data Unit	SETPOS	Reads the last element from the stack (rpn: n), sets the position of the bit stream pointer to this value (bpp: m) and sets the stack pointer to stack position (rpn: n)

2) *GMM - Mobile Identity*

The special information element *Mobile Identity* (GMM protocol [12.]) should dedicate the most sophisticated rules of the *CmdSequence* element.

The Mobile Identity information element provides either the international mobile subscriber identity, IMSI, the temporary mobile subscriber identity, TMSI, the international mobile equipment identity, IMEI or the international mobile equipment identity together with the software version number, IEMISV.

The *Mobile Identity* is a type 4 information element with a minimum length of 3 octets and 11 octets length maximal.

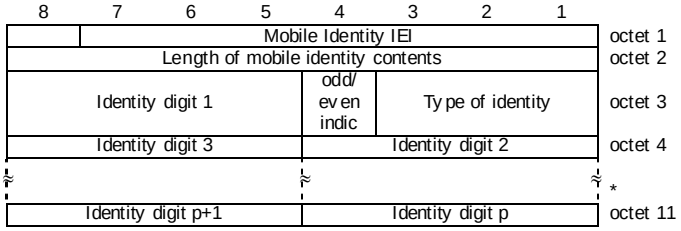


Table 17: Mobile Identity *information element*

The Type of identity (octet 3) determines the kind of identity (IMSI, IMEI, IMEISV, TMSI/TMSI-P or NoIdentity). In case of an even number of identity digits and also when the TMSI/P-TMSI is used the

Odd/even indication is set to **0**; in case of an odd number of identity digits the Odd/even indication is set to **1**. For the IMSI, IMEI and IMEISV the Identity digits field are coded using BCD coding. If the number of identity digits is even then bits 5 to 8 of the last octet shall be filled with an end mark coded as "1111". If the mobile identity is the TMSI/P-TMSI then bits 5 to 8 of octet 3 are coded as "1111" and bit 8 of octet 4 is the most significant bit and bit 1 of the last octet the least significant bit.

The following table shows how to specify this part:

Structured Element Items							
Name	**Pattern**	**Bit Len**	**Type**	**Type Modifier**	**Condition**	**CmdSequence**	**Comment**
Type of identity						GETPOS,:,4,+,:,1,+,SETPOS	Type of identity
Odd/ Even indic						SETPOS	Odd/ Even indicaction
Identity digit			BCDODD	[0..16]	type_of_identity # ID_TYPE_NO_IDENT AND type_of_identity # ID_TYPE_TMSI	SETPOS	Identity digit
	1111	4			type_of_identity = ID_TYPE_TMSI	:,SETPOS,8,+	spare
TMSI				[.32]	type_of_identity = ID_TYPE_TMSI	SETPOS	TMSI or P-TMSI
Dmy				[0..16]	type_of_identity = ID_TYPE_NO_IDENT	SETPOS	dummy

Table 18: SAPE Table Belonging to the Mobile Identity *IE value part*

Explanation:

Type of Identity:	GETPOS	Determines the current bit stream pointer position (bpp: m) and writes this address information to the RPN stack (rpn: n)
	:	Duplicates the address information (rpn: n+1)
	4	Is pushed to the next stack position (rpn: n+2)
	+	Adds the constant value 4 to the bit stream pointer position and saves the result (bpp: m+4) instead of the duplicate of the old bit stream pointer position on the stack (rpn: n+1)
	:	Duplicates the new bit stream pointer position information ((bpp: m+4 -> rpn: n+2)
	1	Is pushed to the next stack position (rpn: n+3)
	+	Adds the constant value 1 to the new bit stream pointer position and saves the result (bpp: m+5) instead of the duplicate of the new bit stream pointer position on the stack (rpn: n+2)
	SETPOS	Reads the last element from the stack (rpn: n+2), sets the position of the bit stream pointer to this value (bpp: m+5) and sets the stack pointer to stack position (rpn: n+1)
Odd/ Even indic	SETPOS	Reads the last element from the stack (rpn: n+1), sets the position of the bit stream pointer to this value (bpp: m+4) and sets the stack pointer to stack position (rpn: n)
In case of ***Type of Identity = TMSI***		
'1111' Spare	:	Duplicates the address information (bpp: m -> rpn: n+1)
	SETPOS	Reads the upper element from the stack (rpn: n+1), sets the position of the bit stream pointer to this value (bpp: m) and sets the stack pointer to stack position (rpn: n+1)
	8	Is pushed to the next stack position (rpn: n+1)
	+	Adds the constant value 8 to the bit stream pointer position and saves the result (bpp: m+8) instead of the original bit stream pointer position on the stack (rpn: n)
TMSI	SETPOS	Reads the upper element from the stack (rpn: n), sets the position of the bit stream pointer to this value (bpp: m+8)
In case of ***Type of Identity = TMSI***		
'1111' Spare	:	Duplicates the address information (bpp: m -> rpn: n+1)
	SETPOS	Reads the upper element from the stack (rpn: n+1), sets the position of the bit stream pointer to this value (bpp: m) and sets the stack pointer to stack position (rpn: n+1)
	8	Is pushed to the next stack position (rpn: n+1)
	+	Adds the constant value 8 to the bit stream pointer position and saves the result (bpp: m+8) instead of the original bit stream pointer position on the stack (rpn: n)
TMSI	SETPOS	Reads the upper element from the stack (rpn: n), sets the position of the bit stream pointer to this value (bpp: m+8)
In case of ***Type of Identity*** = ID_TYPE_NO_IDENT		
Dmy	SETPOS	Reads the upper element from the stack (rpn: n), sets the position of the bit stream pointer to this value (bpp: m)

In all other cases:		
Identity digit	SETPOS	Reads the upper element from the stack (rpn: n), sets the position of the bit stream pointer to this value (bpp: m); The value of the first binary coded decimal digit should be provided as the most significant bits (bit positions 8, 7, 6, 5)

2.2.4.1.4 BitGroupDefinition Element

The BitGroupDef element supports another possibility to specify a certain kind of MsgStructElemItem or MsgItem. SAPE provide a column with the same name as this element. This control element enables a special kind of information elements called *extended octet group*.

At the beginning of each octet there is a flag bit which is set to 1 if the current octet should be followed by a further octet. It is set to 0, if the current octet is the last one in the extended group. These information elements are referred as optional elements.

To specify information elements belonging to an *extended octet group* the first IE of the first octet is associated with the symbol '+' assigned to a BitGroupDef element. The last IE of the last octet is concatenated with the symbol '-' belonging to a BitGroupDef element. The middle IEs situated between information elements marked with '+' and '-' are not joined with any BitGroupDef elements. A single octet of this type is marked with '*' in the BitGroupDef column.

An example of this application is the information element *Cause* (CC protocol [12.]). The following table shows how to specify the value part:

8	7	6	5	4	3	2	1	
	Cause IEI							octet 1
Length of cause contents								octet 2
0/1 ext	coding standard		0 spare	location				octet 3
1 ext	recommendation							octet 3a*
1 ext	cause value							octet 4
diagnostic(s) if any								octet 5*
								octet N*

Table 19: CC Cause *information element*

The purpose of the cause information element is to describe the reason for generating certain messages, to provide diagnostic information in the event of procedural errors and to indicate the location of the cause originator. The cause IE is a GSM4_TLV type information element with a minimum length of 4 octets and a maximum length of 32 octets. If the default value applies to the recommendation field, octet 3a shall be omitted.

The following table shows how to specify this part:

Structured Element Items						
Name	**Pattern**	**Bit Len**	**Type**	**Type Modifier**	**Bit Group Def**	**Comment**
cs					+	Coding standard II
	0	1				Spare
loc					-	Location
rec					*	Recommendation
cause					*	Cause value
diag				[0..27]		Diagnostics

Table 20: SAPE Table Belonging to the CC Cause *IE value part*

Another notation used in the BitGroupDef column describes information elements of an extended group or fields of repetitive elements. Any further octets relating to a protocol extension are marked with '!' or '#'. The last sign marks the end of such an extended group. The example below should point up the

following context: If a received message has less or more extended facsimile capability receiver octets than specified by this table, CCD will skip over the difference (octets or IE number) and will not produce any error report.

Example:

This BCS Digital transmit command message is the response to the standard capabilities identified by the DIS signal (Fax Protocol Entity specified in the ITU-T.30)

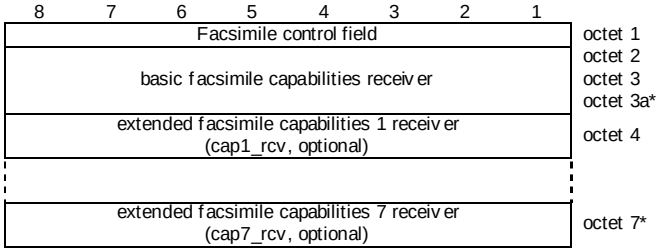


Table 21: BCS Digital transmit command message

The following table shows how to specify this part:

Structured Element Items						
Name	**Pattern**	**Bit Len**	**Type**	**Bit Group Def**	**Presence**	**Comment**
fcf					**MANDATORY**	Facsimile control field
cap0_rcv					**MANDATORY**	basic facsimile capabilities receiver
cap1_rcv				**!**	**OPTIONAL**	extended facsimile capabilities 1 receiver
cap2_rcv				**!**	**OPTIONAL**	extended facsimile capabilities 2 receiver
cap3_rcv				**!**	**OPTIONAL**	extended facsimile capabilities 3 receiver
cap4_rcv				**!**	**OPTIONAL**	extended facsimile capabilities 4 receiver
cap5_rcv				**!**	**OPTIONAL**	extended facsimile capabilities 5 receiver
cap6_rcv				**!**	**OPTIONAL**	extended facsimile capabilities 6 receiver
cap7_rcv				#	**OPTIONAL**	extended facsimile capabilities 7 receiver

Table 22: SAPE Table Belonging to the CC Cause *IE value part*

2.2.4.2 Type

The optional Type element holds information about the type of coding rules needed for the IE. The content must be a valid type definition for the tool chain (e.g. C-types, CCDtypes).

Optional Information Elements (IEs) require coding types either initiated by an identifier (T part) or which must be of one of the following types *CSN.1, S_PADDING* and *GSM5_V*. The latter one has been introduced for parts of a message, which can or must be passed without decoding. Optionality is not only given by coding type. Also conditional IEs are defined as optional for the Coder/Decoder.

The **type** column defines the coding type for the element in question. The **type** column is allowed in air interface message element tables and structured element tables only.

The corresponding file *ccd_codingtypes.h* helps to know which types are supported by which version of Coder/Decoder. Chapter 4 provides some detailed information about supported coding types.

2.2.5 Trivial AIM Specific Sub-Elements

The TI tool chain supplies special coding types to specify arrays of binary coded decimal digits (short: Trivial AIM Specific Sub-Elements).

The intention of these trivial AIM Specific Sub-Elements, which may occur in different context, does not need any further explanation because their names are self-explanatory. They are listed here because of their correct appearance required by the XML schema. The SAPE tool supports the right formatting. But

if an XML document shall be written or modified by using a text editor the user has to be informed how the format must look like.

Element MsgID	
Type	extension of **xs:string**
Used by	element **MsgDef**
Attributes	Name: IDType, Direction; Type: valTypeChoice, xs:string; Use: required, required; Default: ; Fixed:
XML schema	<xs:element name="MsgID"> <xs:complexType> <xs:simpleContent> <xs:extension base="xs:string"> <xs:attribute name="IDType" type="valTypeChoice" use="required"/> <xs:attribute name="Direction" use="required"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="UPLINK"/> <xs:enumeration value="DOWNLINK"/> <xs:enumeration value="BOTH"/> </xs:restriction> </xs:simpleType> </xs:attribute> </xs:extension> </xs:simpleContent> </xs:complexType> </xs:element>
XML example	<MsgID Direction="**UPLINK**" IDType="**DEC**">**1**</MsgID>

Element ItemTag	
Type	extension of **xs:string**
Used by	Elements **MsgItem** **MsgStructElemItem**
Attributes	Name: TagType; Type: valTypeChoice; Use: required; Default: ; Fixed:
XML schema	<xs:element name="ItemTag"> <xs:complexType> <xs:simpleContent> <xs:extension base="xs:string"> <xs:attribute name="TagType" type="valTypeChoice" use="required"/> </xs:extension> </xs:simpleContent> </xs:complexType> </xs:element>
XML example	<ItemTag TagType="**HEX**">**19**</ItemTag>

2.2.6 AIM Specific Attribute Type Definitions

Attributes are used to associate name-value pairs with elements. Often attributes qualify the range of values or determine how to interpret the element's content. The attribute type definitions in this part belong to air interface message specific elements.

SimpleType compTypeChoice	
Type	restriction of **xs:string**
Used by	Attribute **MsgStructElemDef/@Type**
Facets	Enumeration STRUCT Enumeration UNION
XML schema	<xs:simpleType name="compTypeChoice"> <xs:restriction base="xs:string"> <xs:enumeration value="STRUCT"/>

```
    <xs:enumeration value="UNION"/>
  </xs:restriction>
</xs:simpleType>
```

SimpleType valTypeChoice	
Type	restriction of **xs:string**
Used by	Attributes MsgID/@IDType ItemTag/@TagType Value/@ValueType ValuesRange/@ValueType
Facets	Enumeration DEC Enumeration BIN Enumeration HEX Enumeration OCT Enumeration ALPHA
XML schema	<xs:simpleType name="valTypeChoice"> <xs:restriction base="xs:string"> <xs:enumeration value="DEC"/> <xs:enumeration value="BIN"/> <xs:enumeration value="HEX"/> <xs:enumeration value="OCT"/> <xs:enumeration value="ALPHA"/> </xs:restriction> </xs:simpleType>

Presentation Attribute

The SAPE tool provides in the tables *Structured Element Items* and *Message Items* an additional column labelled with the keyword *Presence* to declare the whole item as optional, conditional or mandatory. The XML schema definition handles this presence information by the mandatory attribute *Presentation*, which is concatenated with each item itself. This attribute may take one of the values MANDATORY, OPTIONAL or CONDITIONAL.

The relevant protocol specification may define three different presence requirements (M, C, or O):

- M ("Mandatory") means that the sending side shall include the item, and that the receiver diagnoses a missing mandatory item error when detecting that the item is not present.
- C ("Conditional") means:
  - that inclusion of the item by the sender depends on conditions specified in the relevant protocol specification;
  - that there are conditions for the receiver to expect that the item is present and/or conditions for the receiver to expect that the item is absent in a received message; these conditions depend only on the content of the message itself, and not for instance on the state in which the message was received, or on the receiver characteristics; they are known as static conditions;
- O ("Optional") means that the receiver shall never diagnose a missing mandatory item error because it detects that the item is present or absent.

SimpleType presChoice	
Type	restriction of **xs:string**
Used by	Attributes MsgItem/@Presentation MsgStructElemItem/@Presentation
Facets	Enumeration OPTIONAL Enumeration MANDATORY Enumeration CONDITIONAL
XML schema	<xs:simpleType name="presChoice"> <xs:restriction base="xs:string"> <xs:enumeration value="OPTIONAL"/> <xs:enumeration value="MANDATORY"/> <xs:enumeration value="CONDITIONAL"/> </xs:restriction>

	</xs:simpleType>

2.3 Primitive Specific Part

This part contains a description of the service access point in a specific format (XML), which can be processed by the TI tool chain, resulting in output for the various TI tools (test tools, tracing tools, programming tools etc.). This format is fairly simple and changes to primitives etc are easily done. If the end-result is seen from the viewpoint of a programmer working on implementation of a protocol stack, the end-result corresponds to a set of includable source files containing the definition of the SAP as C declarations. In addition to this SAPE provides a tool for automatic creation of documentation in html format from the original SAP definition document available in xml-format.

One of the great benefits of using the SAP concept is that the resulting code will be structured according to the code standard. This way consistency is ensured in declarations as names of valid flags, counters etc. will have a consistent format throughout the code. This way it will also be easier to read code written by different developers, as the code standard will be kept. Furthermore there will be no name clashes since newer SAP documents can use PREFIX on entity level or alias name on element level. All in all this SAP concept is a single source concept, which allows for both code and documentation in one. That is, it is possible to maintain the documentation and the code at the same time and at the same time ensure consistency in the code.

The aim of this part is to explain how to define service access points. It will describe how the service access point must be structured and how the different elements can be combined.

A Service Access Point XML document is created from a XML schema by using the SAPE tool and choosing SAP (Service Access Point) type when creating a new document. The list below shows the elements belonging to the top level. Some sections may be left out; these sections are marked [optional].

Document Information	Container for all document relevant information
Pragma [optional]	Used to control and to modify the behaviour of the TI tool chain
Constant [optional]	Contains global constants and used to assign a value to a variable
Primitives [optional]	The actual descriptions for all primitive elements belonging to the service access point
Functions [optional]	Functions
Structured Elements [optional]	Container for all Structured Primitive Elements
Basic Elements [optional]	Basic types/values
Substitute [optional]	Used to define a new name for an existing element
Values [optional]	Used as aliases for user specific values
Annotations [optional]	Container for additional information belonging to any document's

Each section above contains a number of subsections, which act as keywords, separating different types of information. Some of these subsections occur in both types of documents (either SAP or AIM) and are already mentioned in part 2.1.

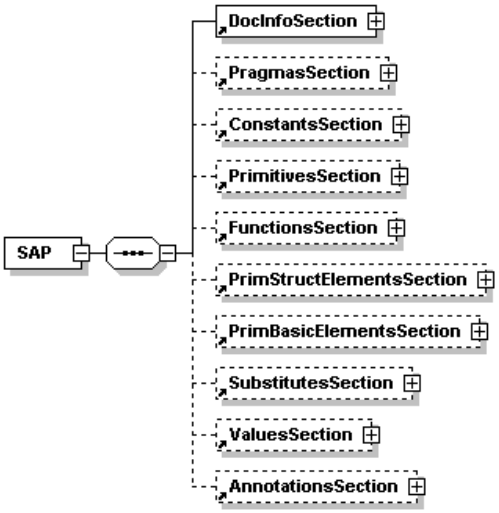


Figure 72: Model of a Service Access Point Description

Element SAP	
Children	DocInfoSection PragmasSection ConstantsSection PrimitivesSection FunctionsSection PrimStructElementsSection PrimBasicElementsSection SubstitutesSection ValuesSection AnnotationsSection
XML schema	<xs:element name="SAP"> <xs:complexType> <xs:sequence> <xs:element ref="DocInfoSection"/> <xs:element ref="PragmasSection" minOccurs="0"/> <xs:element ref="ConstantsSection" minOccurs="0"/> <xs:element ref="PrimitivesSection" minOccurs="0"/> <xs:element ref="FunctionsSection" minOccurs="0"/> <xs:element ref="PrimStructElementsSection" minOccurs="0"/> <xs:element ref="PrimBasicElementsSection" minOccurs="0"/> <xs:element ref="SubstitutesSection" minOccurs="0"/> <xs:element ref="ValuesSection" minOccurs="0"/> <xs:element ref="AnnotationsSection" minOccurs="0"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<SAP xmlns:xsi="**http://www.w3.org/2001/XMLSchema-instance**" xsi:noNamespaceSchemaLocation="**sap.xsd**"> ... PragmasSection ... PragmasSection ... PrimitivesSection ... PrimitivesSectionFunctionsSection ... FunctionsSectionPrimStructElementsSection ... PrimStructElementsSectionPrimBasicElementsSection ... PrimBasicElementsSectionSubstitutesSection ... SubstitutesSection ... AnnotationsSection ... AnnotationsSection

2.3.1 Primitives Section

The *Primitives Section* deals with the Primitive Elements supported by the SAPE editor in table format. It handles the *Primitives Section* element, which comprises **all** primitives declared within an SAP document.

Figure 73: SAPE Messages Section

The *Primitives Section* serves as a container for all primitive elements. The SAPE GUI provides four different parts:

- The Description part should serve as additional information and contains a textual description of the information in the Primitives table. The input corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The table with the name Common Primitive Data manages two *PrimitivesSection's* attributes, which are common to all primitives of a SAP document. One of them is the unique ID for the SAP described by the SAP document (*SAP ID*). Another attribute is called *PrimIDType*. This attribute defines the type of primitive identifier for all primitives of the SAP document. Both attributes are used to associate a primitive tag with an integer primitive identifier (**ID**), which must be unique within the system (cf. Primitive Identifier)
- The SAPE editor offers a table named *Primitives* providing a separate row for each child element. One to many child elements Primitive, which are described in a particular subsection of this document, deal with the internal structure of the Primitive elements. At least one Primitive element has to be present.

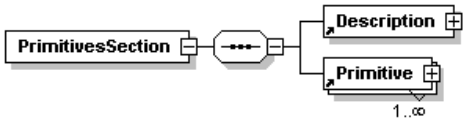


Figure 74: Model of the Messages Section

Element PrimitivesSection	
Children	Description Primitive
Used by	Element SAP
Attributes	Name / Type / Use / Default / Fixed SAPid / xs:string / required PrimIDType / xs:string / required
XML schema	<xs:element name="PrimitivesSection"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="Primitive" maxOccurs="unbounded"/> </xs:sequence> <xs:attribute name="SAPid" type="xs:string" use="required"/> <xs:attribute name="PrimIDType" use="required"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="BIT16"/> <xs:enumeration value="BIT32"/> </xs:restriction> </xs:simpleType> </xs:attribute> </xs:complexType> </xs:element>
XML example gmmreg.sap	<PrimitivesSection PrimIDType="**BIT16**" SAPid="**51**"> ... ... </PrimitivesSection>

2.3.1.1 Primitive

The SAPE GUI provides a table labelled *Primitives* to hold the *Primitive* elements, which comprise all instruments being necessary to define a primitive in a SAP document. The Primitive element serves as a sort of container itself to group a set of information defining its properties. Each row in the Primitives table is associated with another set of tables.

Except the DocLinks table each table relates to an XML element. Table labels and element names can be implicated easily.

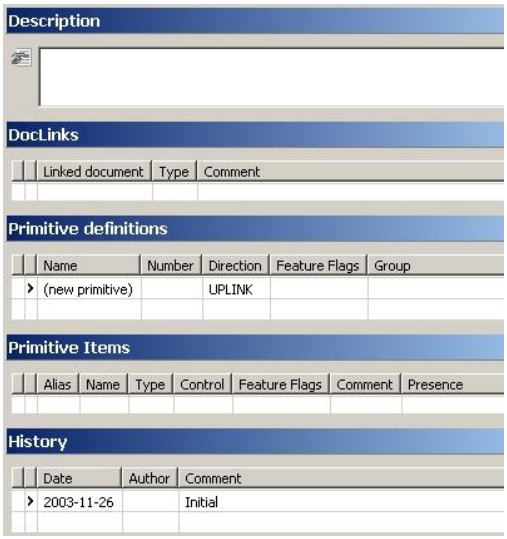


Table 23: SAPE Primitive

The Primitive format should correspond to other key elements in the document. Therefore also Primitive elements have the mandatory elements *Description element* and *History element*. The SAPE GUI offers three parts to take these elements into account:

- The Description part should serve as additional information about each Primitive and corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The History table is intended to track changes. Each row in this table corresponds to one History element. At least one associated XML element is mandatory and therefore a preselection is suggested.

The *Primitive Definitions* table belongs to another mandatory XML element: PrimDef. It acts as a definition element. In case that the layout of the primitive should be assigned to more than only one primitive definition, additional PrimDef elements could be present.

The other table provided by the SAPE GUI relate to an optional XML element.

- The *Primitive Items* table supports one to many PrimItem child elements. These elements are optional and could be attached to each Primitive element if values should be assigned to.

Each of the last two child elements consists of nested elements itself.

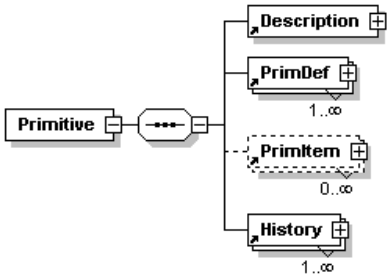


Figure 75: Model of a Primitive Element

Element Primitive	
Children	**Description PrimDef PrimItem History**
Used by	Element **PrimitivesSection**
XML schema	<xs:element name="Primitive"> <xs:annotation> <xs:appinfo> <replaceName PrimDef.Name-=""/> </xs:appinfo> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="PrimDef" maxOccurs="unbounded"/> <xs:element ref="PrimItem" minOccurs="0" maxOccurs="unbounded"/> <xs:element ref="History" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<Primitive> <Description>. . . </Description> <PrimDef> <Name>**GMMREG_ATTACH_REQ**</Name> <PrimID Direction="**UPLINK**" Number="**0**" /> <PrimUsage>. . . </PrimUsage> </PrimDef> <PrimItem Presentation="**MANDATORY**"> <ItemLink> <DocName DocType="**SAP**">**gmmreg**</DocName> <Name>**mobile_class**</Name> </ItemLink> <Comment>**Mobile Class**</Comment> </PrimItem> . . . <PrimItem Presentation="**MANDATORY**">. . . </PrimItem> <History>. . . </History> </Primitive>

2.3.1.1.1 Primitive Definitions

The *Primitive Definitions* table provided by the SAPE GUI holds the PrimDef element, which defines the key parameters of a *Primitive Definition* element in a SAP document.

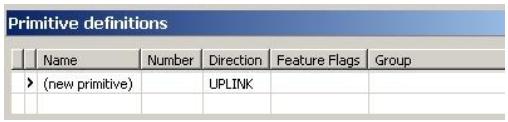


Figure 76: SAPE Primitive Definition Table

The PrimDef element consists of the following child elements:

- The Name element serves as unique identification.
  This element is mandatory and could be used to reference this *Primitive Element* by other elements. It corresponds to a column in the *Primitive Definitions* table with the same name. The Name should follow the TI coding standard. A primitive may for instance be a request (_REQ), a confirmation (_CNF), an indication (_IND) or a response (_RES).
- The PrimID) element is associated with two attributes, which are used to generate an integer primitive identifier (**ID**). This primitive identifier must be unique within the system (cf. Primitive Identifier). In the *Primitive Definitions* table there are two columns (named *Number* and *Direction*) to keep these attributes' values.
- The optional Version element
  accepts any combination of text or digits that represents the dependency from feature flags for

the definition of a specific message element. This element relates to the Feature Flags column. The entries are optional but must comply with coding rules for feature flags.

- The PrimUsage element itself consists of the mandatory child elements *Sender* and *Receiver* in order to specify the sending and receiving entities. If more than one entity sends/receives the primitive on the SAP, additional direction specifications can be present. Therefore one to many *PrimUsage* elements are allowed. For every sender/receiver pair, a *PrimUsage* element has to be created. The SAPE editor provides a nested sub-table to handle the *PrimUsage* elements. A double dick to a name entry in the *Primitive Definitions* table will open this nested table.

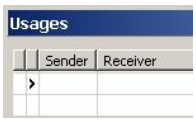


Figure 77: SAPE Primitive Usage Table

- The Group element (optional)
could be present to declare a group where the *Primitive Element* belongs. Groups can be used to force the generators of the tool chain to separate the definitions of elements into different output files. Any combination of text or digits is allowed. A separate column in the *Primitive Definitions* table with the same name corresponds to this child element.

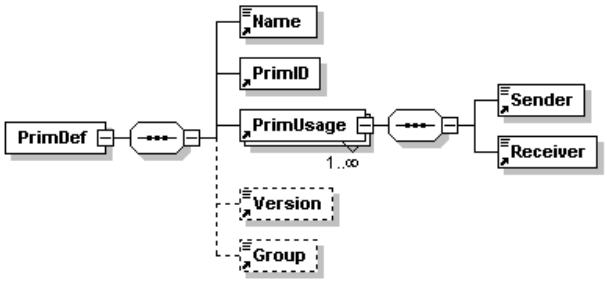


Figure 78: Model of a Primitive Definition Element

Element PrimDef	
Children	**Name PrimID PrimUsage Version Group Comment**
Used by	Element **Primitive**
XML schema	<xs:element name="PrimDef"> <xs:complexType> <xs:sequence> <xs:element ref="Name"/> <xs:element ref="PrimID"/> <xs:element name="PrimUsage" maxOccurs="unbounded"/> <xs:complexType> <xs:sequence> <xs:element name ="Sender" type="xs:string" <xs:element name="Receiver" type="xs:string" </xs:sequence> </xs:complexType> </xs:element> <xs:element name ="Version" type="xs:string" minOccurs="0"/> <xs:element name="Group" type="xs:string" minOccurs="0"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<PrimDef> <Name>**GMMREG_ATTACH_REQ**</Name> <PrimID Direction="**UPLINK**" Number="**0**" /> <PrimUsage> <Sender>**MMI**</Sender> <Receiver>**GMM**</Receiver> </PrimUsage> </PrimDef>

2.3.1.1.2 Primitive Items

The *Primitive Items* table provided by the SAPE GUI holds the PrimItem elements (sub-elements that will be transported by the primitive), which define the items of a primitive element. A primitive is always seen as a user defined structural type, since it is created for the purpose of passing data from one entity to another. The effect is the same as defining a parameter of content type **STRUCT**. Consequently, the definition also corresponds to a type declaration in C: A new structural type is declared with a body defined by the "elements" part of the section.

Figure 79: SAPE Primitive Items Table

The PrimItem element comprises all components which are needed to declare a primitive item. A PrimItem is a reference to an existing SAP or AIM element. A complex PrimItem element could be a composition of well-assorted references. All elements have to be defined explicitly. To reference an existing item located in the same or an external document, the *ItemLink* element will be used.

Each row of the *Primitive Items* table provided by the SAPE GUI corresponds to a primitive item (PrimItem). The columns of the *Primitive Items* table diverge a little bit from the strict conversion to support exactly one column for each child element.

The PrimItem element consists of the child elements:

- To reference an existing item located in the same or an external document, the ItemLink element will be used. It represents a reference to an item defined elsewhere in the same or an external document. The SAPE column labelled with the keyword *Name* offers the possibility to join a linked item by reference. The SAPE editor provides the possibility to select a new linked element from the Repository Entry and to jump to the linked element. The type column in the *Primitive Items* table will be filled automatically: SAPE recognizes the type of the linked item

(e.g. U8 belonging to a certain Basic Primitive Element or STRUCT in case of an arbitrary Structured Primitive Element). This column serves as user-friendly information part, but it does not relate to an independent child element of the PrimItem element.

- The Alias element (optional)
  enables any combination of text or digits to identify a user specific value. This element could be used to define the new name in the case that the name of the item, under which it could be addressed within the primitive, should be different from the name of the linked element. By default the SAPE table doesn't provide this column, but it can be switched on easily by selecting the appropriate keyword in the *Visible Optional Columns* field.
- With the Control element (optional)
  the item could be modified (e.g. array, dependencies, conditionals). The SAPE tool provides in the *Structured Element Items* table separate column for the control sub element. The behaviours of the different control mechanisms are described in a separate part below.
- The optional Version element
  accepts any combination of text or digits that represents the dependency from feature flags for the definition of a specific message element. This element relates to the Feature Flags column. The entries are optional but must comply with coding rules for feature flags.
- The Comment element (mandatory)
  enables any combination of text or digits that should make a comment. Ideally it should not be the repetition of the comment, where the element was defined; it should describe the usage of the element within the primitive element. The associated XML element is mandatory and therefore a preselection is suggested.

The SAPE tool provides in the *Primitive Items* table an additional column labelled with the keyword *Presence* to declare the whole primitive item as optional or mandatory. The XML schema definition handles this presence information by the mandatory attribute Presentation, which is concatenated with each PrimItem. This attribute may take one of the values MANDATORY or OPTIONAL. If there exists a Primitive Item at least one component is needed to declare this item. Therefore a preselection is suggested in the *Presence* column.

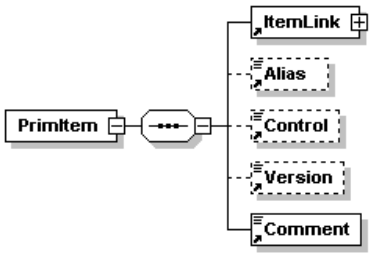


Figure 80: Model of a Message Item Element

Element PrimItem	
Children	**ItemLink Alias Control Version Comment**
Used by	Element **Primitive**
Attributes	Name: Presentation; Type: presChoice; Use: required; Default: ; Fixed:

XML schema	`<xs:element name="PrimItem">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="ItemLink"/>`
	`<xs:element ref="Alias" minOccurs="0"/>`
	`<xs:element ref="Control" minOccurs="0"/>`
	`<xs:element ref="Version" minOccurs="0"/>`
	`<xs:element ref="Comment"/>`
	`</xs:sequence>`
	`<xs:attribute name="Presentation" type="presChoice" use="required"/>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<PrimItem Presentation="MANDATORY">`
	`<ItemLink>`
	`<DocName DocType="SAP">gmmreg</DocName>`
	`<Name>mobile_class</Name>`
	`</ItemLink>`
	`<Comment>Mobile Class</Comment>`
	`</PrimItem>`

2.3.2 Structured Elements Section

The *Structured Elements Section* deals with the Structured Primitive Elements supported by the SAPE editor in table format. It handles the PrimStructElementsSection element, which comprises **all** structured elements declared within an SAP document.

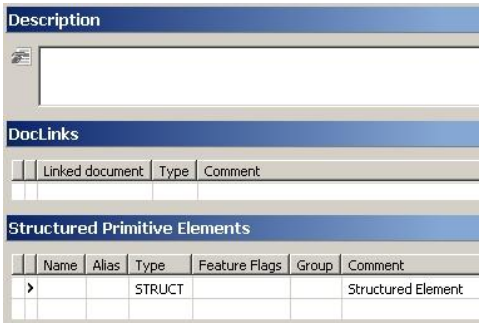


Figure 81: SAPE Structured Primitive Elements Section

This section provided by SAPE contains three different parts:

- The Description part should serve as additional information about the Structured Elements Section and should contain a textual description of the information in the Structured Primitive Elements table below. This part corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The third part is a table to take one to many child elements PrimStructElem (*Structured Primitive Elements* - mandatory) into account. They are described in a particular subsection of this document that deals with the internal structure of the structured primitive elements. At least one PrimStructElem element has to be present. The SAPE editor offers a table labelled *Structured Primitive Elements* providing a separate row for each child element PrimStructElem.

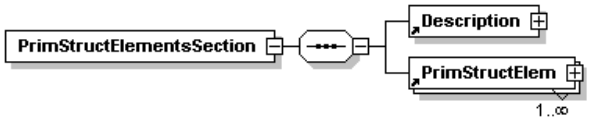


Figure 82: Model of the Primitive Structured Element Section

Element PrimStructElementsSection	
Children	**Description** **PrimStructElem**
Used by	Element **SAP**
XML schema	<xs:element name="PrimStructElementsSection"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="PrimStructElem" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<PrimStructElementsSection> <Description>...</Description> <PrimStructElem>...</PrimStructElem> ... <PrimStructElem>...</PrimStructElem> </PrimStructElementsSection>

2.3.2.1 Structured Primitive Elements

Each row of the table labelled *Structured Primitive Elements* in the Structured Elements Section represents a separate child element called PrimStructElem . By selecting any child element respectively any row the SAPE GUI provides a set of tables needed to describe a single element sufficiently. These tables comprise all instruments being necessary to define a structured element in an SAP document. Each table relates to an XML element. Table labels and element names can be implicated easily.

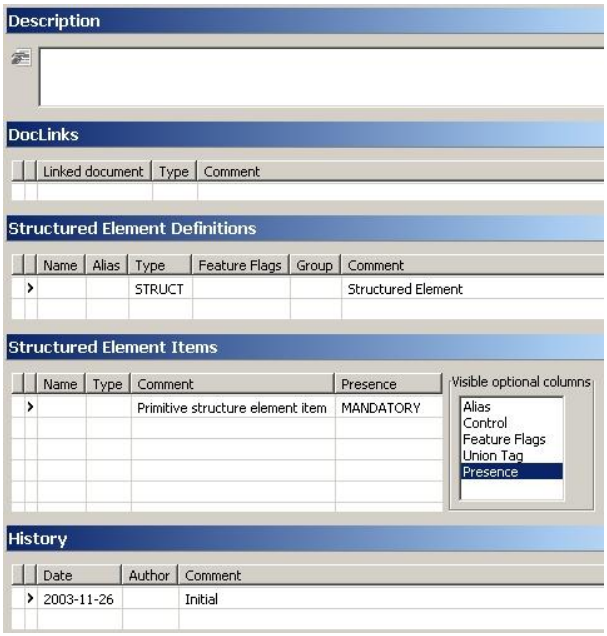


Figure 83: SAPE Structured Primitive Elements Tables

The PrimStructElem element serves as a sort of container itself to group a set of information defining its properties. Each row in the *Structured Primitive Elements* table mentioned above is associated with another set of tables. The PrimStructElem format corresponds to other key elements in the document. Therefore also PrimStructElem elements have the mandatory elements *Description element* and *History element*.

The SAPE GUI offers three parts to take these elements into account:

- The Description part should serve as additional information about the PrimStructElem element and corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The History table is intended to track changes. Each row in this table corresponds to one History element. At least one associated XML element is mandatory and therefore a preselection is suggested.

Additional tables are provided:

- The *Structured Primitive Element Definitions* table (short: PrimStructElemDef element), which is mandatory, act as a definition element. In case that the layout of the structured element should be assigned to more than only one element definition, additional PrimStructElemDef elements could be present.
- The *Structured Primitive Element Items* table supports one to many PrimStructElemItem child elements. These elements are optional. They could be attached to the PrimStructElem element if values should be assigned to the structured element.

Each of these child elements consists of nested elements itself.

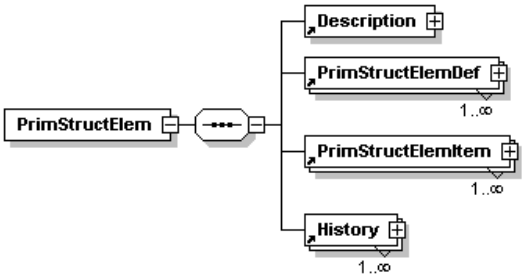


Figure 84: Model of a Structured Primitive Element

Element PrimStructElem	
Children	**Description** **PrimStructElemDef** **PrimStructElemItem** **History**
Used by	Element **PrimStructElementsSection**
XML schema	<xs:element name="PrimStructElem"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="PrimStructElemDef" maxOccurs="unbounded"/> <xs:element ref="PrimStructElemItem" maxOccurs="unbounded"/> <xs:element ref="History" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<PrimStructElem> <Description>...</Description> <PrimStructElemDef Type="**STRUCT**">>...</PrimStructElemDef> <PrimStructElemItem Presentation="**MANDATORY**>...</</PrimStructElemItem> ... <PrimStructElemItem Presentation="**MANDATORY**>...</</PrimStructElemItem> <History>...</History> </PrimStructElem>

2.3.2.1.1 Structured Primitive Element Definitions

The *Structured Element Definition* table provided by the SAPE GUI holds the PrimStructElemDef element, which defines the key parameters of a simple element in an SAP document.

Figure 85: SAPE Structured Element Definitions Table

Each column in the *Structured Element Definition* table corresponds to an XML child element of the PrimStructElemDef element. The column labels are equivalent to the child element names.

The PrimStructElemDef element consists of the child elements:

- The Name element serves as unique identification.
  This element is mandatory and could be used to reference this *Structured Element* by other elements.
- The Alias element (optional)
  enables any combination of text or digits to identify a user specific value. The SAPE GUI offers

a separate column to support text input. This element holds an alias name for a PrimStructElem element.

- The optional Version element
  accepts any combination of text or digits that represents the dependency from feature flags for the definition of a specific message element. This element relates to the Feature Flags column. The entries are optional but must comply with coding rules for feature flags.
- The Group element
  could be present to declare a group where the *Structured Primitive Elements* belong. Groups can be used to force the generators of the tool chain to separate the definitions of elements into different output files. Any combination of text or digits is allowed.
- The Comment element (mandatory)
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory and therefore a preselection is suggested.

The SAPE tool provides in the *Structured Element Definitions* table an additional column to set the type of each PrimStructElem element. The XML schema definition handles this type information by the mandatory attribute *Type*. This attribute may take either the value STRUCT or the value UNION and will typically result in an item according to the C programming language.

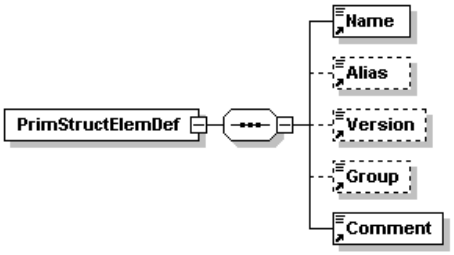


Figure 86: Model of a Structured Primitive Element Definition

Element PrimStructElemDef					
Children	**Name Alias Version Group Comment**				
Used by	Element **PrimStructElem**				
Attributes	Name Type	Type compType-Choice	Use required	Default	Fixed
XML schema	<xs:element name="PrimStructElemDef"> <xs:complexType> <xs:sequence> <xs:element ref="Name"/> <xs:element ref="Alias" minOccurs="0"/> <xs:element name ="Version" type="xs:string" minOccurs="0"/> <xs:element name="Group" type="xs:string" minOccurs="0"/> <xs:element name ="Comment type="xs:string""/> </xs:sequence> <xs:attribute name="Type" type="compTypeChoice" use="required"/> </xs:complexType> </xs:element>				
XML example	<PrimStructElemDef Type="**STRUCT**"> <Name>**plnm**</Name> <Comment>**PLMN identification**</Comment> </PrimStructElemDef>				

2.3.2.1.2 Structured Primitive Element Items

The *Structured Element Items* table provided by the SAPE GUI holds the PrimStructElemItem elements, which define the items of a structured or combined element. The PrimStructElemItem element

comprises all components which are needed to declare a structured message element item. Basically a PrimStructElemItem could be a reference to an existing element. A complex PrimStructElemItem element could be a composition of well-assorted references.

Each row of the *Structured Element Items* table provided by the SAPE GUI corresponds to a single structured element item (PrimStructElemItem). The columns of the *Structured Element Items* table diverge a little bit from the strict conversion to support exactly one column for each child element.

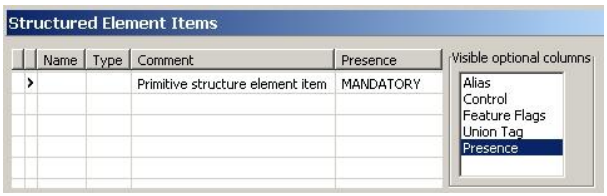


Figure 87: SAPE Structured Primitive Element Items Table

The child elements of the PrimStructElemItem element are declared below.

- A structured primitive element item is a reference to an existing element. All items will be declared using an ItemLink element.

  To reference an existing item located in the same or an external document, the ItemLink element will be used. It represents a reference to an item defined elsewhere in the same or an external document. The SAPE column labelled with the keyword *Name* offers the possibility to join a linked item by reference. The SAPE editor provides the possibility to select a new linked element from the Repository Entry and to jump to the linked element.

- The Alias element (optional)
  enables any combination of text or digits to identify a user specific value. This element could be used to define the new name in the case that the name of the item, under which it could be addressed within the primitive, should be different from the name of the linked element. By default the SAPE table doesn't provide this column, but it can be switched on easily by selecting the appropriate keyword in the *Visible Optional Columns* field.

- The Type element (optional)
  holds information about the type of an item. The content must be a valid type definition for the tool chain (e.g. C-types, CCDtypes). Any combination of text or digits is allowed.

- The UnionTag element (optional)
  is a tag identifier of union elements that indicates which union element of all possible elements is present.

- With the Control element (optional)
  the item could be modified (e.g. array, dependencies, conditionals). The SAPE tool provides in the *Structured Element Items* table a separate column for the control sub element, which can be switched to visible or turned off. The behaviours of the different control mechanisms are described in a separate part below.

- The optional Version element
  accepts any combination of text or digits that represents the dependency from feature flags for the definition of a specific message element. This element relates to the Feature Flags column. The entries are optional but must comply with coding rules for feature flags.

- The Comment element (mandatory)
  enables any combination of text or digits that should make a comment. Ideally it should not be the repetition of the comment, where the element was defined; it should describe the usage of the element within the structured element. The associated XML element is mandatory and therefore a preselection is suggested.

The SAPE tool provides in the *Structured Element Items* table an additional column labelled with the keyword *Presence* to declare the whole structured element item as optional or mandatory. The XML

schema definition handles this presence information by the mandatory attribute Presentation, which is concatenated with each PrimStructElemItem. This attribute may take one of the values MANDATORY or OPTIONAL. If there exists a *Structured Element Item* at least one component is needed to declare this item. Therefore a preselection is suggested in the *Presence* column.

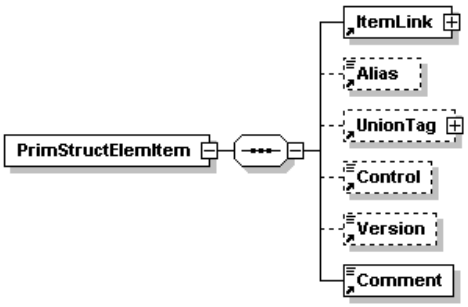


Figure 88: Model of a Structured Primitive Element Item

Element PrimStructElemItem	
Children	ItemLink Alias UnionTag Control Version Comment
Used by	Element PrimStructElem
Attributes	Name: Presentation; Type: presChoice; Use: required; Default: ; Fixed:
XML schema	<xs:element name="PrimStructElemItem"> <xs:complexType> <xs:sequence> <xs:element ref="ItemLink"/> <xs:element ref="Alias" minOccurs="0"/> <xs:element ref="UnionTag" minOccurs="0"/> <xs:element ref="Control" minOccurs="0"/> <xs:element name ="Version" type="xs:string" minOccurs="0"/> <xs:element name ="Comment" type="xs:string"/> </xs:sequence> <xs:attribute name="Presentation" type="presChoice" use="required"/> </xs:complexType> </xs:element>
XML example	<PrimStructElemItem Presentation="**MANDATORY**"> <ItemLink> <DocName DocType="**SAP**">**gmmreg**</DocName> <Name>**mcc**</Name> </ItemLink> <Control>**[SIZE_MCC]**</Control> <Comment>**mobile country code**</Comment> </PrimStructElemItem>

2.3.3 Basic Elements Section

The *Basic Element Section* deals with the *Basic Primitive Elements* table provided by the SAPE editor. It handles the PrimBasicElementsSection element, which comprises **all** basic elements declared within an SAP document.

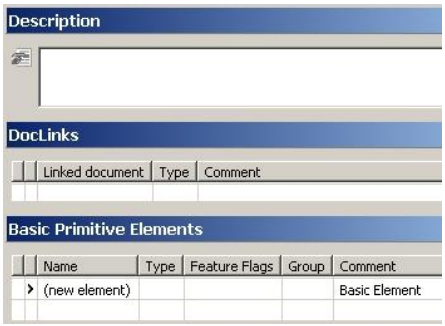


Figure 89: SAPE Basic Elements Section

The PrimBasicElementsSection serves as a container for all basic elements. For these purpose the SPAE GUI provides three different parts:

- The Description part should serve as additional information about the Basic Elements Section and should contain a textual description of the information in the Basic Primitive Elements table below. This part corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The third part is a table to take one to many child elements PrimBasicElem (*Basic Primitive Elements* - mandatory) into account. They are described in a particular subsection of this document, which deals with the internal structure of the Basic Element Definitions. At least one

PrimBasicElem element has to be present. The SAPE editor offers a table labelled *Basic Primitive Elements* providing a separate row for each child element PrimBasicElem.

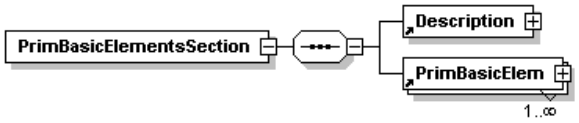


Figure 90: Model of the Basic Element Section

Element PrimBasicElementsSection	
Children	**Description PrimBasicElem**
Used by	Element **SAP**
XML schema	<xs:element name="PrimBasicElementsSection"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="PrimBasicElem" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	<PrimBasicElementsSection> <Description>...</Description> <PrimBasicElem>...</PrimBasicElem> ... <PrimBasicElem>...</PrimBasicElem> </PrimBasicElementsSection>

2.3.3.1 Basic Primitive Elements

Each row of the table labelled *Basic Primitive Elements* in the Basic Elements Section represents a separate child element called PrimBasicElem. By selecting any child element respectively any row the SAPE GUI provides a set of tables needed to describe a single element sufficiently. These tables comprise all instruments being necessary to define a structured element in an SAP document. Each table relates to an XML element. Table labels and element names can be implicated easily.

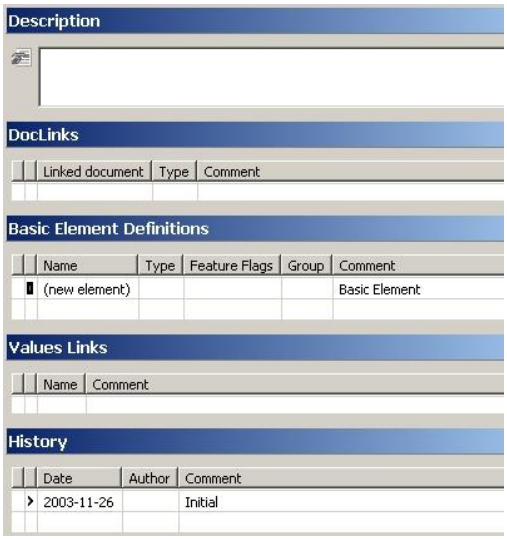


Figure 91: SAPE Basic Primitive Elements Tables

The PrimBasicElem element serves as a sort of container itself to group a set of information defining its properties. Each row in the *Basic Primitive Elements* table mentioned above is associated with another set of tables. The PrimBasicElem format corresponds to other key elements in the document. Therefore also PrimBasicElem elements have the mandatory elements Description element and History element.

The SAPE GUI offers three parts to take these elements into account:

- The Description part should serve as additional information about the PrimBasicElem element and corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The History table is intended to track changes. Each row in this table corresponds to one History element. At least one associated XML element is mandatory and therefore a preselection is suggested.

Additional tables are provided:

- The Basic Element Definitions table (short: PrimBasicElemDef element), which is mandatory, act as a definition element.
- The ValuesLink table supports one to many *Values Link* child elements. These elements are optional. They could be attached to the PrimBasicElem element if values should be assigned to the basic element. This element is optional.

Each of these child elements consists of nested elements itself.

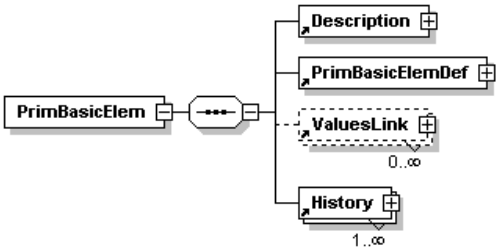


Figure 92: Model of a Basic Primitive Element

Element PrimBasicElem	
Children	**Description PrimBasicElemDef ValuesLink History**
Used by	Element **PrimBasicElementsSection**

XML schema	`<xs:element name="PrimBasicElem">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="Description"/>`
	`<xs:element ref="PrimBasicElemDef"/>`
	`<xs:element ref="ValuesLink" minOccurs="0" maxOccurs="unbounded"/>`
	`<xs:element ref="History" maxOccurs="unbounded"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<PrimBasicElem>`
	`<Description>...</Description>`
	`<PrimBasicElemDef>`
	`<Name>attach_type</Name>`
	`<Type>U8</Type>`
	`<Comment>Attach type</Comment>`
	`</PrimBasicElemDef>`
	`<ValuesLink>`
	`<DocName DocType="SAP">gmmreg</DocName>`
	`<Name>VAL_attach_type</Name>`
	`</ValuesLink>`
	`<History>...</History>`
	`</PrimBasicElem>`

2.3.3.1.1 Basic Element Definitions

The *Basic Element Definitions* table provided by the SAPE GUI holds the PrimBasicElemDef element, which defines the key parameters of a simple element in an SAP document.

Each column in *Basic Element Definitions* table corresponds to an XML child element of the PrimBasicElemDef element. The column labels are equivalent to the child element names - except of the Feature Flags column.

Figure 93: SAPE Basic Element Definitions Table

The PrimBasicElemDef element consists of the following child elements:

- The Name element serves as unique identification.
  This element is mandatory and could be used to reference this *Basic Element* by other elements.

- The Type element holds information about the type of an item. The content must be a valid type definition for the tool chain (e.g. C-types like *U8, S8, U16, S16, U32* or *S32* or BYTE, UBYTE, WORD, UWORD, LONG and ULONG) indicates that the element is a basic one.
- The optional Version element
  accepts any combination of text or digits that represents the dependency from feature flags for the definition of a specific primitive element. This element relates to the Feature Flags column. The entries are optional but must comply with coding rules for feature flags.
- The Group element
  could be present to declare a group where the *Basic Element* belongs. Groups can be used to force the generators of the tool chain to separate the definitions of elements into different output files. Any combinations of text or digits are allowed.
- The Comment element (mandatory)
  enables any combination of text or digits that should make a comment. The associated XML element is mandatory and therefore a preselection is suggested.

A link element is not foreseen in this format, because an element could be either defined or referenced. Therefore a link is only possible where an element will be used.

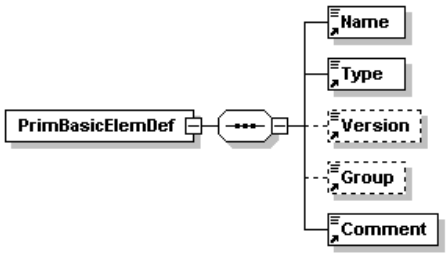


Figure 94: Model of a Basic Element Definition

Element PrimBasicElemDef	
Children	**Name Type Version Group Comment**
Used by	Element **PrimBasicElem**

XML schema	`<xs:element name="PrimBasicElemDef">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="Name"/>`
	`<xs:element ref="Type"/>`
	`<xs:element name ="Version" type="xs:string" minOccurs="0"/>`
	`<xs:element name ="Group" type="xs:string" minOccurs="0"/>`
	`<xs:element name ="Comment" type="xs:string" type="xs:string"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<PrimBasicElemDef>`
	`<Name>attach_type</Name>`
	`<Type>U8</Type>`
	`<Comment>Attach type</Comment>`
	`</PrimBasicElemDef>`

2.3.4 Functions Section

The *Functions Section* deals with the Function Elements supported by the SAPE editor in table format. It handles the FunctionsSection element, which comprises **all** function elements declared within an SAP document. At least one function has to be declared within the *Functions Section*, which may have one to many child elements Function.

The Functions Section enables the generation of function prototypes in the C-Header files during processing SAP specification documents by the Generic Tool Chain. These function prototypes establish the name of the function, its return type, as well as the type and number of its formal parameters. For example this section enables the basics to define functional interfaces of protocol stack entities.

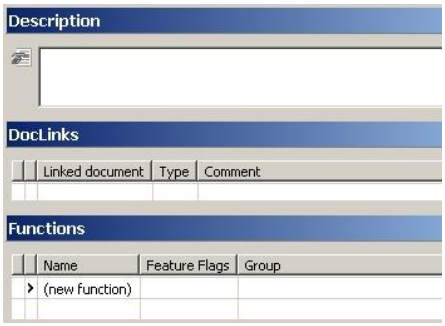


Figure 95: SAPE Functions Section

This section provided by SAPE contains three different parts:

- The Description part should serve as additional information about the *Functions Section* and should provide more information about the functions of that SAP in general. This part corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The third part (mandatory) is a table to take one to many child elements Function into account. They are described in a particular subsection of this document that deals with the internal structure of the function elements. At least one Function element has to be present. The SAPE editor offers a table labelled *Functions* providing a separate row for each child element Function.

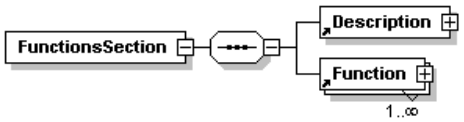


Figure 96: Model of the Functions Sections

Element FunctionsSection	
Children	**Description Function**
Used by	Element **SAP**
XML schema	<xs:element name="FunctionsSection"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="Function" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	[TBD]

2.3.4.1 Functions

The *Function* element comprises all elements which are needed to declare a function in a SAP document. Each row of the table labelled *Functions* in the FunctionsSection represents a separate child element called Function. By selecting any child element respectively any row the SAPE GUI provides a set of tables needed to describe a single element sufficiently. These tables comprise all instruments being necessary to define an inline function in an SAP document. Each table relates to an XML element. Table labels and element names can be implicated easily.

Figure 97: SAPE Function

The Function format corresponds to other key elements in the document. Therefore also Function elements have the mandatory elements Description and History.

The SAPE GUI offers three parts to take these elements into account:

- The Description part should serve as additional information about the Function and corresponds to the child Section of the Description element.
- To link arbitrary files, which are entirely informational, there exists a separate table labelled with *DocLink*. This table is associated with the sub element Linked Description Elements belonging to the parent element Description.
- The History table is intended to track changes. Each row in this table corresponds to one History element. At least one associated XML element is mandatory and therefore a preselection is suggested.

Additional tables are provided:

- The *Usage* table (short: FuncUsage element) belongs to mandatory child elements from the FuncDef element. The caller/callee relationship of the function will be presented by the FuncUsage element. If more than one usage relationship can be defined, additional FuncUsage

elements can be attached to the FuncDef element. Each row identifies the involved entities (e.g. a certain caller/callee pair).

- The Function Definition table (short: FuncDef element) is mandatory and acts as an element defining the key parameters of a function in a SAP document.
- The *Arguments* table comprises all function arguments. Each row belongs to a FuncArg element, which defines an argument of a function. The presence of the FuncArg elements is optional because functions do not need to be declared with arguments. One to many FuncArg elements may be present.
- The Return Values table provides the possibility to define one return value of a function. As well function declarations without a return value are allowed, therefore the dedicated FuncRet element is optional.

Each of these child elements consists of nested elements itself.

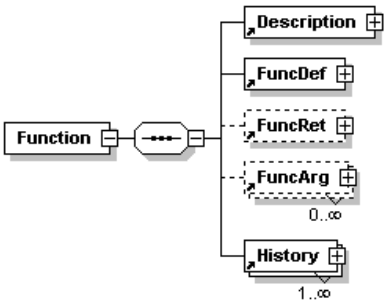


Figure 98: Model of a Function Element

Element Function	
Children	**Description FuncDef FuncRet FuncArg History**
Used by	Element **FunctionsSection**
XML schema	<xs:element name="Function"> <xs:complexType> <xs:sequence> <xs:element ref="Description"/> <xs:element ref="FuncDef"/> <xs:element ref="FuncRet" minOccurs="0"/> <xs:element ref="FuncArg" minOccurs="0" maxOccurs="unbounded"/> <xs:element ref="History" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	[TBD]

2.3.4.1.1 Function Definitions

The Function Definitions table (short: FuncDef element) provided by SAPE consists of three columns:

- The Name column,
  which entries are mandatory, could be used to reference this function.
- The Feature Flags column
  relates to the *Version* element which offers alphanumerical data fields and represents the dependency from feature flags for a specific item. The entries are optional but must comply with coding rules for feature flags.
- The Group column
  offers alphanumerical data field, too. These optional entries can hold the name of a group.

Groups can be used to force the generators of the tool chain to separate the definitions of elements into different output files. This column relates to an XML element of the same name.

Figure 99: SAPE Function Definition Table

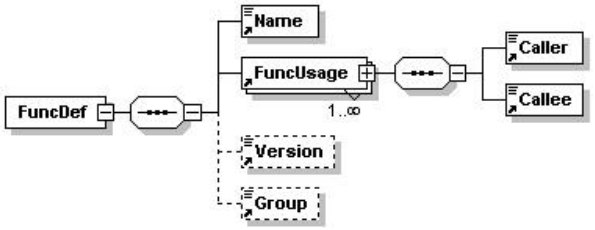


Figure 100: Model of a Function Definition Element

The FuncDef element has one to many child elements called FuncUsage. Each element specifies a certain caller/callee relationship of the function. If more than one usage relationship can be defined, additional FuncUsage elements can be attached to the FuncDef element. AT least one FuncUsage child elements must be present.

Element FuncDef	
Children	**Name FuncUsage Version Group**
Used by	Element **Function**

XML schema	`<xs:element name="FuncDef">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="Name"/>`
	`<xs:element name ="FuncUsage" maxOccurs="unbounded"/>`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element name="Caller" type="xs:string"/>`
	`<xs:element name="Callee" type="xs:string"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`
	`<xs:element name ="Version" type="xs:string" minOccurs="0"/>`
	`<xs:element name="Group" type="xs:string" minOccurs="0"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	[TBD]

2.3.4.1.2 Function Arguments

The *Arguments* table provided by the SAPE GUI holds the FuncArg elements, which comprises all components needed to declare a function argument.

Figure 101: SAPE Function Arguments Table

Each row of the *Arguments* table corresponds to a single function argument. The columns of this table diverge a little bit from the strict conversion to support exactly one column for each child element.

The FuncArg element consists of the following child elements:

- It is only possible to declare a function argument by a reference to an existing SAP or AIM element or to reference an externally defined type. All elements have to be defined explicitly. To reference an existing item located in the same or an external document, the ItemLink element will be used. It represents a reference to an item defined elsewhere in the same or an external document. The SAPE column labelled with the keyword *Name* offers the possibility to join a linked item by reference. The SAPE editor provides the possibility to select a new linked element from the Repository Entry and to jump to the linked element.

  The ExtType element will be used if a type is needed that is out of the scope of SAP and AIM Documents. Therefore the Type information and optionally an ExtSource element have to be defined as child elements. It is mandatory to choice one of these alternatives.

- The Alias element (optional)
  enables any combination of text or digits to identify a user specific value. The SAPE GUI offers a separate column to support text input. This element holds an alias name for a FuncArg element in case the name of the item, under which it could be addressed within the argument list, should be different from the name of the linked element
- With the Control element (optional)
  the item could be modified (e.g. array, pointer). The behaviours of the different control mechanisms are described in a separate part below.
- The Comment element (mandatory)
  enables any combination of text or digits that should make a comment. Ideally it should not be the repetition of the comment, where the element was defined; it should describe the usage of the element as a function argument.

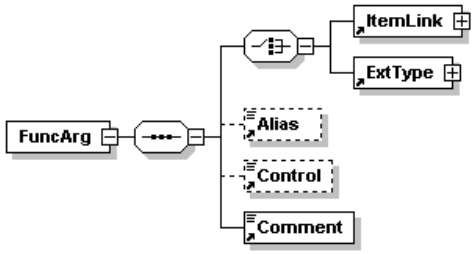


Figure 102: Model of a Function Argument Element

Element FuncArg	
Children	**ItemLink ExtType Alias Control Comment**
Used by	Element **Function**
XML schema	<xs:element name="FuncArg"> <xs:complexType> <xs:sequence> <xs:choice> <xs:element ref="ItemLink"/> <xs:element ref="ExtType"/> </xs:choice> <xs:element ref="Alias" minOccurs="0"/> <xs:element ref="Control" minOccurs="0"/> <xs:element ref="Comment"/> </xs:sequence> </xs:complexType> </xs:element>
XML example	[TBD]

2.3.4.1.3 Function Return Value

The *Return Value* table provided by the SAPE GUI holds the FuncRet elements, which comprises all components needed to declare a function return value.

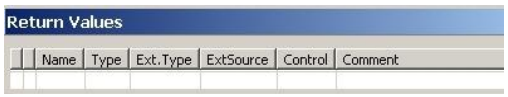


Figure 103: SAPE Function Arguments Table

The Return Values table provides the possibility to define a single return value of a function. The columns of this table diverge a little bit from the strict conversion to support exactly one column for each child element.

The FuncRet element consists of the following child elements:

- It is only possible to declare a function return value by a reference to an existing SAP or AIM element or to reference an externally defined type. All elements have to be defined explicitly. To reference an existing item located in the same or an external document, the ItemLink element will be used. It represents a reference to an item defined elsewhere in the same or an external document. The SAPE column labelled with the keyword *Name* offers the possibility to join a linked item by reference. The SAPE editor provides the possibility to select a new linked element from the Repository Entry and to jump to the linked element.

  The ExtType element will be used if a type is needed that is out of the scope of SAP and AIM Documents. Therefore the Type information and optionally an ExtSource element have to be defined as child elements. It is mandatory to choice one of these alternatives.

- With the Control element (optional)
  the item could be modified (e.g. array, pointer). The behaviours of the different control mechanisms are described in a separate part below.

- The Comment element (mandatory)
  enables any combination of text or digits that should make a comment. Ideally it should not be the repetition of the comment, where the element was defined; it should describe the usage of the element as a function return value.

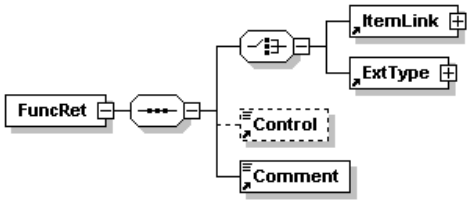


Figure 104: Model of a Function Return Value Element

Element FuncRet	
Children	**ItemLink ExtType Control Comment**
Used by	Element **Function**

XML schema	`<xs:element name="FuncRet">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:choice>`
	`<xs:element ref="ItemLink"/>`
	`<xs:element ref="ExtType"/>`
	`</xs:choice>`
	`<xs:element ref="Control" minOccurs="0"/>`
	`<xs:element name="Comment" type="xs:string"/> </xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	[TBD]

2.3.5 Nontrivial SAP Specific Sub-Elements

Some SAP specific sub-elements, which may occur in different context, require more detailed explanation. These elements are listed here to provide additional information about the data they may contain. This section should serve primarily as a reference for these elements.

2.3.5.1 Primitive Identifier

For primitives the definition associates a primitive tag with an integer primitive identifier (**ID**), which must be unique within the system. This corresponds to defining a global constant using the "#define" pre-processor directive in C, linking the primitive name to the value of the **ID**.

The name of the primitive is contained in the tabular definition under the heading **Name**; the global primitive identifier numerical value (**ID**) is assembled by

- the unique ID for the SAP (*SAP ID*),
- a numerical identifier for that primitive given by the Number attribute and
- the direction information given.

The SAPE tool provides in the *Primitive Definitions* table a particular column to set the direction information for each *Primitive Element*. The XML schema definition handles this direction information by the mandatory attribute *Direction*[10], which belongs to the mandatory Primitive Identifier element. This attribute may take one of the possible values UPLINK or DOWNLINK. The *Direction* attribute indicates the information flow direction according to the ETSI/3GPP specifications:

UPLINK - An "uplink" is a unidirectional radio link for the transmission of signals from a UE towards a core network.

DOWNLINK - A “downlink” is a unidirectional radio link for the transmission of signals from a core network towards a UE.

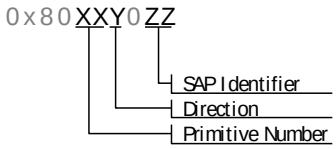
[10] Besides the direction information is an important entry for the coder/decoder.

The Number attribute enables any combination of text or digits, but it is recommended to use positive integer values. The primitive number is the number of the primitive within the SAP and should start at 00 for the first primitive in the document.

For primitives of type *Request* or *Response*, the direction should have a value of 0. For primitives of type *Confirm* or *Indication* the direction should have a value of 4. The SAPE converter cares for the correct number according to the given direction information.

The numerical value of the primitive identifier should be a 32 bit unsigned integer, but older SAP descriptions use 16 bit primitive IDs. Therefore the SAPE converter must be advised which value representation is appropriate. The attribute *PrimIDType,* which is common to all primitives of a SAP document and belongs to the *PrimitivesSection*, performs that task.

The scheme below shows the guidelines to assemble a 32-bit ID using the hexadecimal format:



Element PrimID	
Used by	Element **PrimDef**
Attributes	Name / Type / Use / Default / Fixed Number / xs:string / required Direction / xs:string / required

XML schema	`<xs:element name="PrimID">`
	`<xs:complexType>`
	`<xs:attribute name="Number" type="xs:string" use="required"/>`
	`<xs:attribute name="Direction" use="required">`
	`<xs:simpleType>`
	`<xs:restriction base="xs:string">`
	`<xs:enumeration value="UPLINK"/>`
	`<xs:enumeration value="DOWNLINK"/>`
	`</xs:restriction>`
	`</xs:simpleType>`
	`</xs:attribute>`
	`</xs:complexType>`
	`</xs:element>`

XML example	`<PrimID Direction="UPLINK" Number="0" />`

2.3.5.2 Control

The Control element belonging to SAP descriptions is not as complex as the AIM Control element. The SAP Control possibilities are a subset of the AIM Control facilities. It contains instructions used by the TI tool chain how to define or interpret the associated object in question.

This optional element is intended to hold any kind of control information to classify an item. With the information contained in the Control element an item could be modified in order to achieve different constructions, such as arrays and pointers. The SAPE tool provides in the Structured Primitive Element Items table and in the Primitive Items table as far as in the Function Arguments and Function Return Value table a separate column for the control element, which can be switched to visible or turned off.

In contradiction to AIM Control elements no classification is done.

Element Control	
Type	**xs:string**
Used by	elements **FuncArg FuncRet PrimItem PrimStructElemItem**
XML schema	<xs:element name="Control" type="xs:string"/>
XML example	<Control>**[SIZE_MNC]**</Control>

2.3.5.2.1 Element Arrays

Arrays can only be used in the elements part of a declaration of a primitive or a parameter with the content type **STRUCT**.

Note: Arrays of unions are not supported.

Each item in a table being an array needs its own length specifier to define the number of elements in the array. All elements of one array have the same content type. Different kinds of arrays can be distinguished: An array may have a fixed number of elements or a variable number of elements. The syntax definition of the length specifier is given below.

Syntax Definition:

TypeModifier ::= "["MinimumElementNumber [".." MaximumElementNumber] "]" | "0.." MaximumElementNumber "]"

MinimumElementNumber ::= Constant [ArithmOp Number]

MaximumElementNumber ::= Constant [ArithmOp Number]

Constant ::= Number | ConstantAlias

Number ::= (NonZeroNum {Num})

NonZeroNum ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Num ::= **0** | NonZeroNum

ConstantAlias ::= /* Reference to an Alias element of the Constants part */

ArithmOp ::= **+** | **-** | ***** | ****

Arrays with a fixed number of elements need the *MinimunElementNumber* only. This value can be a positive number of integral types or the Alias of global constants with a value larger than zero.

Structured Element Items / Primitive Items			
Name	**Type**	**Control**	**Comment**
mnc	U8	[SIZE_MNC]	mobile network code

Table 24: Example of an Array with Fixed Length

For fixed size arrays the array specification for an element results in the C expression:

```
T_SHORT_NAME <name>[<MinimumElementNumber>];
```

For basic types, no user-defined type is used, and **the content type specified for the element replaces T_SHORT_NAME**.

Arrays with a variable number of elements need an upper limit named *MaximumElementNumber* and a lower limit, which is either zero or a *MinimumElementNumber* and must be less than the *Maximum-ElementNumber*. Each limit can be a positive number of integral types or the Alias of a global constant.

Structured Element Items / Primitive Items			
Name	**Type**	**Control**	**Comment**
text	U8	[0..MMR_MAX_TEXT_LEN]	name

Table 25: Example of an Array with Variable Length

Comment [K8]: Unions must be encapsulated in a structure in order to create an array. The structure requirement is due to the extra union controller ("ctrl_") element inserted by the TI tool chain; this element is outside the union, and thus needs a structure to contain it. ->Additional investigation needed: SAPE handles this problem as if unions are structures. But how does the TI toll chain work?

A specification of an array with a variable number of elements results in a slightly different construction of the C generated:

```
U8 <c_short_name>;
T_SHORT_NAME <short_name> [<MaximumElementNumber>];
```

The parameter c_short_name is a counter providing information about the number of elements in the variable size array. The array is actually declared of maximum size, and only the array elements **[0 ; c_short_name – 1]** contain valid data.

2.3.5.2.2 Element Pointers

Pointers can be used as elements in declarations of primitives, functions and parameters of content type **STRUCT** or **UNION**. Pointers are made possible by addition of entries in the Control column belonging to the table in the elements declaration part. For each element in the table that should be a pointer, the keyword **PTR** is added in the **Control** column, causing generation of a C construction shown below:

```
T_SHORT_NAME * <ptr_short_name>;
```

short_name is valid for both the element itself and the parameter definition referred.

If the element is of content type **UNION**, the C construction will look like the example below:

```
T_CTRL_SHORT_NAME <ctrl_short_name>;       /* Comment */
T_SHORT_NAME * <ptr_short_name>;           /* Comment */
```

This construction allows the identification of the element chosen in the union pointed to.

Pointers are primarily used in order to share storage between the entities using a common SAP, passing only the references to data. The memory will however need to be allocated independently of the SAP. When using the PTR keyword it is very important to be aware of the consequences. One extremely important point is that the entire store pointed to is traced out. This may result in insufficient bandwidth when testing and therefore should be used with care.

2.3.5.2.3 Dynamic Arrays

Particular Control elements offer the possibility to create dynamic size arrays. The keywords **DYN** or **PTR** allow the specification of these arrays. The amount of memory is dynamically allocated dependent on the number of elements. Dynamic size arrays (where either the keyword **DYN** or **PTR** is used) are constructed similar to arrays with a variable number of elements.

The keyword **DYN** in the Control column will cause the dedicated C-code shown below (code transparent):

```
U8 c_short_name;
T_SHORT_NAME * short_name;
```

If the keyword **PTR** is used the dedicated C-code will look like the example below (non code transparent):

```
U8 c_short_name;
T_SHORT_NAME * ptr_short_name;
```

In both cases the parameter **c_short_name** is a counter providing information about the number of elements in the array. The C examples show that specifications of dynamic size arrays using either the keyword **DYN** or the keyword **PTR** cause a similar behavior. In both cases the result is the generation of a pointer to the specified type. The elements can be accessed in the same way as elements of fixed or variable size arrays, since the C syntax will be the same. **DYN** and **PTR** declarations differ only in the naming of the pointer.

The main difference between dynamic size arrays and fixed or variable size arrays is the way of memory allocation. The pointer references the memory allocated for the array, which will have the size appropriate to hold the number of entries indicated by **c_short_name**.

Declarations using either the keyword **DYN** or the keyword **PTR** differ in the behavior of optional elements. If the keyword **PTR** is used relating to an optional element - either for a dynamic size array or for a pointer – no valid flag is added. In case of missing this optional the element the value of the pointer is set to NULL. If the keyword **DYN** is used relating to an optional element a valid flag is added.

Please note that it is not possible to make array of pointers (cf. part 2.2.4.1.1)

2.3.5.3 Extern Type

The *ExtType* element will be used if a type is needed that is out of the scope of SAP and AIM Documents. The mandatory child element Type is described in a separate part below The optional child element ExtSource enables alphanumerical data and provides to hold the name of an external source file, which must be included to resolve an external type.

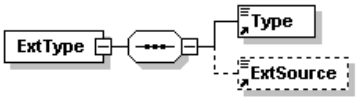


Figure 105: Model of an External Type Element

Element ExtType	
Children	**Type ExtSource**
Used by	Elements **FuncArg FuncRet**

XML schema	`<xs:element name="ExtType">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element ref="Type"/>`
	`<xs:element name="ExtSource" type="xs:string" minOccurs="0"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

XML example	[TBD]

2.3.5.4 Type

This plain element provides information about the type of an item. Any combination of text or digits is allowed, but the content must be a valid type definition for the tool chain (e.g. C-types, CCDtypes).

Element Type	
Type	**xs:string**
Used by	Elements **ExtType PrimBasicElemDef**
XML schema	`<xs:element name="Type" type="xs:string"/>`
XML example	`<Type>U8</Type>`

2.3.6 SAP Specific Attribute Type Definitions

Presentation Attribute

The SAPE tool provides in the tables *Structured Primitive Element Items* and *Primitive Items* an additional column labelled with the keyword *Presence* to declare the whole item as optional or mandatory. The XML schema definition handles this presence information by the mandatory attribute Presentation, which is concatenated with each item itself. This attribute may take one of the values MANDATORY or OPTIONAL.

For each element, a presence specifier is to be added. . For mandatory elements, this has no consequence for the C construction, but for optional elements the result is:

```
U8 v_short_name;
T_SHORT_NAME short_name;
```

A valid flag v_short_name is added to the construction. The value of the parameter v_short_name indicates whether the content of the element is valid or not. The value can be 0 for not valid or 1 for valid. If the keyword **PTR** is used relating to an optional element - either for a dynamic size array or for a

pointer – no valid flag is added. In case of missing this optional the element the value of the pointer is set to NULL. If the keyword **DYN** is used relating to an optional element a valid flag is added.

SimpleType presChoice	
Type	restriction of **xs:string**
Used by	Attributes **PrimItem/@Presentation PrimStructElemItem/@Presentation**
Facets	Enumeration OPTIONAL Enumeration MANDATORY Enumeration CONDITIONAL
XML schema	<xs:simpleType name="presChoice"> <xs:restriction base="xs:string"> <xs:enumeration value="OPTIONAL"/> <xs:enumeration value="MANDATORY"/> <xs:enumeration value="CONDITIONAL"/> </xs:restriction> </xs:simpleType>

3 Message and Primitive Editorial Description Catalogues - Microsoft Word documents

3.1 Message Specific Part

[Refer to [3.]: TI User Guide - Syntax description for air interface message documents, 8350_300_MSG_Syntax.doc]

3.2 Primitive Specific Part

[Refer to [4.]: TI User Guide - Specifying Service Access Points, 8350_301_SAP_Syntax.doc]

4 Coding Types

CCD uses particular functions to encode/decode the different types and formats of information elements. The following part provides an overview of the most important coding types used for GSM/GPRS. Coding type names used in CCD differ just a little from those in message description. For example the type GSM1_V will change into CCDTYPE_GSM1_V in CCD source code. Until now CCD has considered the following coding types:

1. For standard information elements:
   GSM1_V, GSM1_TV, GSM2_T, GSM3_T, GSM3_TV, GSM4_LV, GSM4_TLV, GSM5_V, GSM5_TV, GSM5_TLV, GSM1_ASN and GSM1_ASN_NULL.
2. For non-standard information elements:
   BCDODD, BCDEVEN, BCD_MNC, BCD_NOFILL, T30_IDENT, CSN1_S1, CSN1_S0, CSN1_SHL, S_PADDING, S_PADDING_0, GSM6_TLV, GSM7_LV, NO_CODE, CSN1_CONCAT, BREAK_COND, CSN1_CHOICE1 and CSN1_CHOICE2.

The corresponding file *ccd_codingtypes.h* helps to know which types are supported by which version of Coder/Decoder. Chapter provides some detailed information about supported coding types.

It is possible to use several coding types within a mixed or nested Information element.

Type names used in *.xml and in *.mdf files	In ccd_codingtypes.h	Source file name	
GSM1_V	CCDTYPE_GSM1_V	gsm_1v.c	Valid types for GSM and GPRS
GSM1_TV	CCDTYPE_GSM1_TV	gsm1_tv.c	
GSM2_T	CCDTYPE_GSM2_T	gsm2_t.c	
GSM3_V	CCDTYPE_GSM3_V	gsm3_v.c	
GSM3_TV	CCDTYPE_GSM3_TV	gsm3_tv.c	
GSM4_LV	CCDTYPE_GSM4_LV	gsm4_lv.c	
GSM4_TLV	CCDTYPE_GSM4_TLV	gsm4_tlv.c	
GSM5_V	CCDTYPE_GSM5_V	gsm5_v.c	
GSM5_TV	CCDTYPE_GSM5_TV	gsm5_tv.c	
GSM5_TLV	CCDTYPE_GSM5_TLV	gsm5_tlv.c	
GSM6_TLV	CCDTYPE_GSM6_TLV	gsm6_tlv.c	
GSM7_LV	CCDTYPE_GSM7_LV	gsm7_lv.c	
GSM1_ASN	CCDTYPE_GSM1_ASN	gsm1_asn.c	
BCDODD	CCDTYPE_BCDODD	bcdodd.c	
BCDEVEN	CCDTYPE_BCDEVEN	bcdeven.c	
BCD_NOFILL	CCDTYPE_BCD_NOFILL	bcd_nofill.c	
BCD_MNC	CCDTYPE_BCD_MNC	bcd_mnc.c	
CSN1_S1	CCDTYPE_CSN1_S1	csn1_s1.c	
CSN1_SHL	CCDTYPE_CSN1_SHL	csn1_shl.c	
S_PADDING	CCDTYPE_S_PADDING	s_padding.c	
T30_IDENT	CCDTYPE_T30_IDENT	t30_ident.c	
BITSTRING	CCDTYPE_BITSTRING	asn1_bitstr.c	Valid types used for elements of UMTS messages
ASN1_OCTET	CCDTYPE_ASN1_OCTET	asn1_octet.c	
ASN1_INTEGER	CCDTYPE_ASN1_INTEGER	asn1_integ.c	

ASN1_SEQUENCE	CCDTYPE_ASN1_SEQUENCE	asn1_seq.c	
ASN1_CHOICE	CCDTYPE_ASN1_CHOICE	asn1_choice.c	
NO_CODE			
ASN1_INTEGER_EXTENSIBLE			
S_PADDING_0	CCDTYPE_S_PADDING_0	s_padding_0.c	
CSN1_S0	CCDTYPE_CSN1_S0	csn1_s0.c	
HL_FLAG	CCDTYPE_HL_FLAG	hl_flag.c	
FDD_CI	CCDTYPE_FDD_CI	fdd_ci.c	
TDD_CI	CCDTYPE_TDD_CI	tdd_ci.c	
FREQ_LIST	CCDTYPE_FREQ_LIST	freq_list.c	
CSN1_CONCAT	CCDTYPE_CSN1_CONCAT	csn1_concat.c	
BREAK_COND	CCDTYPE_BREAK_COND	break_cond.c	
CSN1_CHOICE1	CCDTYPE_CSN1_CHOICE1	csn1_choice_1.c	
CSN1_CHOICE2	CCDTYPE_CSN1_CHOICE2	csn1_choice_2.c	

Table 26: Currently valid types of coding rules

4.1 Coding Types for Standard Information Elements

A standard L3 message consists of an imperative part, itself composed of a header and the rest of imperative part, followed by a non-imperative part. Both the non-header part of the imperative part and the non-imperative part are composed of successive parts referred as standard information elements.

Imperative Part		Non-Imperative Part
msg header	first IE . . . last IE	first IE . . . last IE

Table 27: Scheme of a Standard Information Element

CCD does not handle the message header since it is processed by the GSM protocol entities. Hence a description of concepts like Protocol Discriminator, Skip Indicator and Transaction Identifier does not needed here.

A standard IE may have the following parts, in that order:

- an information element identifier (IEI);
- a length indicator (LI);
- a value part.

A standard IE has one of the formats shown in the following table:

Format	Meaning	IEI present	LI present	Value part present
T	Type only	yes	no	no
V	Value only	no	no	yes
TV	Type and Value	yes	no	yes
LV	Length and Value	no	yes	yes
TLV	Type, Length and Value	yes	yes	yes

Table 28: Formats of Information Elements

The information element type describes the meaning of the value part. Standard IEs of the same information element type (IEI) may appear with different formats. The format used for a given standard IE in a given message is specified within the description of the message.

When present, the IEI of a standard IE consists of a half octet or one octet. A standard IE with IEI consisting of a half octet has format TV, and its value part consists of a half octet. The value of the IEI depends on the standard IE, not on its information element type. The IEI, if any, of a given standard IE in a given message is specified within the description of the message. In some protocol specifications, default IEI values can be indicated. They are to be used if not indicated in the message specification. Non mandatory standard IE in a given message, i.e., IE which may be not be present (formally, for which the null string is acceptable in the message), must be formatted with an IEI, i.e., with format T, TV or TLV.

When present, the LI of a standard IE consists of one octet. It contains the binary encoding of the number of octets of the IE value part. The length indicator of a standard IE with empty value part indicates 0 octets. Standard IE of an information element type such that the possible values may have different values must be formatted with a length field, i.e., LV or TLV.

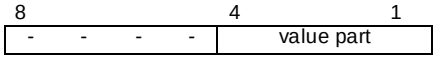
The value part of a standard IE either consists of a half octet or one or more octets; the value part of a standard IE with format LV or TLV consists of an integral number of octets, between 0 and 255 inclusive; it then may be empty, i.e., consist of zero octets; if it consists of a half octet and has format TV, its IEI consists of a half octet, too. The value part of a standard IE may be further structured into parts, called fields.

Categories of IEs; order of occurrence of IEI, LI, and value part

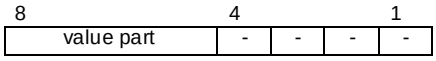
Totally four categories of standard information elements are defined:

1. Information elements of format V or TV with value part consisting of 1/2 octet (type 1)
   They are known as **GSM1_V** and **GSM1_TV** in CCD tables.

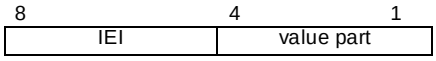
   Type 1 IE of Format **V**
   providing the value in bit positions 8, 7, 6, 5 of an octet

   Type 1 IE of Format **V**
   providing the value in bit positions 4, 3, 2, 1 of an octet

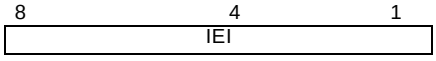
   

   Type 1 IE of Format **TV**
   having an IEI of a half octet length; they provide the IEI in bit positions 8, 7, 6, 5 of an octet and the value part in bit positions 4, 3, 2, 1 of the same octet

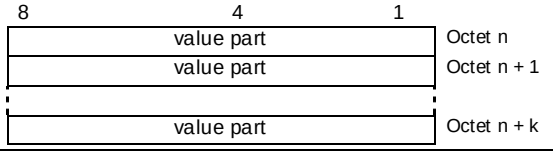
2. Information elements of format **T** with value part consisting of 0 octets (type 2)
   They are known as **GSM2_T** in the CCD tables

   Type 2 IE of Format **T**
   Its IEI consists of one octet, its value part is empty

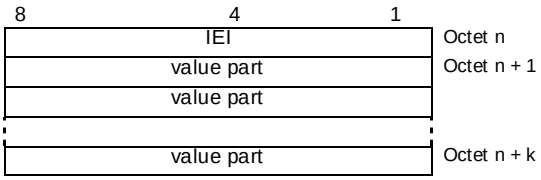
   

3. Information elements of format **V** or **TV** with value part that has fixed length of at least one octet (type 3)
   They are known as GSM3_V and GSM3_TV in the CCD tables

   Type 3 IE of Format **V**
   The value part consists of at least one octet

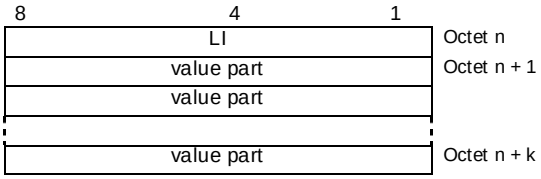
   

Type 3 IE of Format **TV**
It's IEI consists of one octet and proceeds the value part in the IE. The value part consists of at least one octet

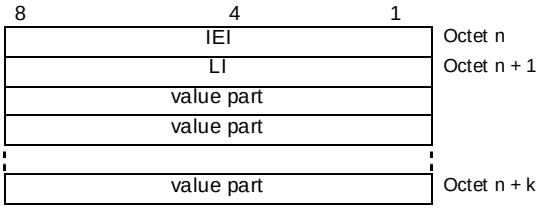


4. Information elements of format **TLV** or **LV** with value part consisting of zero, one or more octets (type 4)
They are known as **GSM4_TV**, **GSM4_TLV**, **GSM1_ASN** and **GSM1_ASN_NULL** in the CCD tables. If present, its IEI has one octet length.

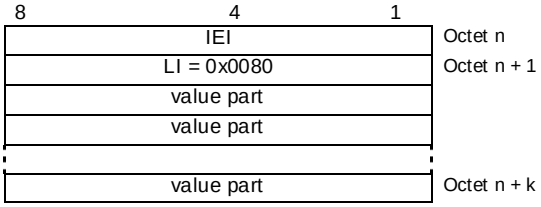
Type 4 IE of Format **LV**
Its LI precedes the value part, which consists of zero, one, or more octets



Type 4 IE of Format **TLV** or **ASN.1 BER**
Its LI precedes the value part, which consists of zero, one, or more octets. If present, its IEI has one octet length and precedes the LI



Type 4 IE of Format **ASN.1 BER** with EOF (End Of Content octets)



In case of elements encoded with ASN.1 BER the LI can be used to signal an unknown length for the value part. If so then the LI is 0x0080. The end of the value part is earmarked by two octets filled with zeros. This is known as indefinite form.

Besides infinite form of length encoding there are also a short and a long form according to BER.

Definite Short Form (L<128):

The length is given in one octet, representing a range of numbers from 0 to 127 since the first bit must be 0. For example, a length field of 01010110 indicates that the content field has 86 octets.

Definite Long Form:

The first bit of the first byte is set to 1. The bottom seven bits (#6-#0) of the first byte indicate the number of bytes of length data to follow. The first subsequent byte is the most significant byte. It is also permitted to insert all-zero bytes between the first byte and the actual length data bytes. Example: 0x81 0x80 means L=128 and is equivalent to 0x81 0x00 0x80.

There are some extensions to support specific GSM/GPRS/UMTS protocol in an easy manner.

- CCD defines a fifth type of Information elements of format **TLV** with value part consisting of zero, one or more octets. It is an extension of GSM4_TLV and is called **GSM5_TLV**. This type supports length identifiers (LI) up to two octets for non ASN.1 BER information elements. If the length information exceeds 127 bytes the first byte of the LI field is dedicated to the constant number 0x81. The second one then contains the length information. The T part (used as information element identifier) consists of an octet.
- Another extended type is called **GSM5_V**. It is used to write raw or undecoded bits, which have been read previously from a received message. The structure of this IE is very easy and the coding is much simpler than for GSM3_V and GSM1_V. The usage of this type in conjunction with the type modifer element **bit array** will increase more efficient definitions.
- The type **GSM6_TLV** is intended to specify elements consisting of an 8 bit T component, a fixed sized 16 Bit L component and a V component which length depends on the value of the L component.
- The TI tool chain provides an additional coding type **GSM7_LV** to de-/encode elements consisting of a 7 bit length component followed by a value componet. The length component determines the bit length of the value component.

4.2 Coding Types for Non-Standard Information Elements

4.2.1 BCD Coding Types

The TI tool chain supplies special coding types to specify arrays of binary coded decimal digits (short: BCD). These coding types are in relationship with the internal order of the array elements, which affect the position within an octet. If the value of the first binary coded decimal digit should be provided as the most significant bits (bit positions 8, 7, 6, 5) the coding type **BCDODD** must be used. The coding type **BCDEVEN** supports arrays of binary coded decimal digits if the value of the first binary coded decimal digit should be provided as the least significant bits (bit positions 4, 3, 2, 1).

The BCD coding must be associated with a Type Modifier Element. This Type Modifier Element determines the total length of the bit field: **Note**: The Type Modifier Element must not be the instruction to build a bit array.

An example of this application is the information element *TP Validity Period (Absolute Format)* (SMS protocol [13.]) shown by the following table:

8	7	6	5	4	3	2	1	
year digit 2				year digit 1				octet 1
month digit 2				month digit 1				octet 2
day digit 2				day digit 1				octet 3
hour digit 2				hour digit 1				octet 4
minute digit 2				minute digit 1				octet 5
second digit 2				second digit 1				octet 6
tz_sign	tz_msb			tz_lsb				octet 7

Table 29: Application of BCDEVEN - TP Validity Period (Absolute Format) *information element*

Structured Element Items			
Name	**Type**	**Type Modifier**	**Comment**
year	BCDEVEN	[2]	Year, two-digit
month	BCDEVEN	[2]	Month, two-digit
day	BCDEVEN	[2]	Day, two-digit
hour	BCDEVEN	[2]	Hour, two-digit
minute	BCDEVEN	[2]	Minute, two-digit
second	BCDEVEN	[2]	Second, two-digit
tz_lsb			Time Zone, LSB
tz_sign			Time Zone, sign
tz_msb			Time Zone, MSB

Table 30: SAPE Table Belonging to TP Validity Period (Absolute Format) *information element*

In the example above the bit field has a constant and known length. In some other cases the length can vary between lower und upper limits. The sub element *num* of the information element *Called party BCD number* (CC protocol [12.]) is an array with a variable number of elements. The first *num* element should be coded as the least significant bits of the first octet. The following table shows how to specify this part:

Structured Element Items			
Name	**Type**	**Type Modifier**	**Comment**
num	BCDEVEN	[0..32]	Number digit (0..32)

Table 31: SAPE Table Belonging to TP Validity Period (Absolute Format) *information element*

If the value part of a Mobile Identity information element belongs to the international mobile subscriber identity, IMSI, the overall number of digits in IMSI shall not exceed 15 digits.

8	7	6	5	4	3	2	1	
Identity digit 1				odd/ even indic	Type of identity			octet 1
Identity digit 3				Identity digit 2				octet 2
≈				≈			≈	*
Identity digit p+1				Identity digit p				octet 9

Table 32: IMSI value

In case of an even number of identity digits the odd/even indication is set to **0**; in case of an odd number of identity digits the odd/even indication is set to **1**. The Identity digits field are coded using BCD coding. If the number of identity digits is even then bits 5 to 8 of the last octet shall be filled with an end mark coded as "1111". The coding type **BCDODD** fulfils these conditions.

The following table shows how to specify this part:

Structured Element Items							
Name	**Pattern**	**Bit Len**	**Type**	**Type Modifier**	**Condition**	**CmdSequence**	**Comment**
Type of identity						GETPOS,:,4,+,:,1,+,SETPOS	Type of identity
Odd/ Even indic						SETPOS	Odd/ Even indicaction
Identity digit			BCDODD	[0..15]		SETPOS	Identity digit

Table 33: SAPE Table Belonging to the IMSI value

Special cases require byte arrays supporting a Mobile Country Code (MCC) element in conjunction with a Mobile Network Code (MNC) element. In accordance with the ETSI / 3gpp specifications coding of the Mobile Network Code field is the responsibility of each administration but BCD coding shall be used. The MNC shall consist of 2 or 3 digits.

An example of this application is the value part of the information element *Location Area Identification* (RR protocol [14.]) which length is 5 bytes. Table 34 shows the internal construction if MNC consists of 2 digits, only. Table 35 belongs to an example of a three-digit MNC.

8	7	6	5	4	3	2	1	
MCC digit 2				MCC digit 1				octet n-2
1	1	1	1	MCC digit 3				
MNC digit 2				MNC digit 1				octet n
LAC								
LAC (continued)								octet n+2

Table 34: Location Area Identification *information element - two-digit MNC*

8	7	6	5	4	3	2	1	
MCC digit 2				MCC digit 1				octet n-2
MNC digit 3				MCC digit 3				
MNC digit 2				MNC digit 1				octet n
LAC								
LAC (continued)								octet n+2

Table 35: Location Area Identification *information element – three-digit MNC*

In case of MCC the number of the digits is odd and the first element should be coded as the least significant bits of the first octet. The last octet must be filled in the most significant nibble either with the bit pattern 1111 or with binary coded MNC digit 3. The first element of MNC should be coded as the least significant bits of the first octet. If MNC consists of 2 digits the number of the digits is even; in case of a three-digit MNC the number of the digits is odd.

The XML specification needs to advise the TI tool chain how to deal with these arrays MCC (Mobil Country Code) and MNC (Mobile Network Code). Therefore two special coding types are provided. **BCD_MNC** supports the unusual position of the MNC digit 3 and **BCD_NOFILL** enables the appropriate interpretation of the most significant nibble in octet (n-1).

The following table shows the completion of the SAPE table:

Structured Element Items			
Name	**Type**	**Type Modifier**	**Comment**
mcc	BCD_NOFILL	[3]	Mobile Country Code
Mnc	BCD_MNC	[2..3]	Mobile Network Code

Table 36: SAPE Table Belonging to Location Area Identification *information element*

Element Type	
Type	**xs:string**
Used by	elements **MsgItem MsgStructElemItem**
XML schema	<xs:element name="Type" type="xs:string"/>
XML example	<Type>**BCD_MNC**</Type>

4.2.2 CSN1 Coding

Some technical specifications written by ETSI/3GPP define structures of messages, which are non-standard L3 messages as defined in [11.] 3GPP TS 24.007. E.g. [15.] 3GPP TS44.060 uses many definitions using CSN.1 descriptions of the message information elements and fields (Description of CSN.1 see 3GPP TS 24.007 and CSN.1 Specification ver. 2.0 Cell&Sys).

In these structures you will find a lot of optional or conditional elements. Notes have to specify the conditions for information elements or fields with presence requirement C or O in the relevant message which together with other conditions define when the information elements shall be included or not, what non-presence of such information elements or fields means, and the static conditions for presence

and/or non-presence of the information elements or fields. Very often a flag represented by a single 1-bit string indicates the presence of optional elements:

Example: <something> ::= 0 | 1 <element>;

Therefore the TI tool chain provides some special coding types to enhance performance. Without these coding types message description would become very complex and the en-/decoding process would require much more time. These coding types help to improve the efficiency of the TI tool chain.

4.2.2.1 CSN1_S1

The CSN1_S1 coding type is an appropriate solution if an optional information element is described by the following example:

<something> ::= 0 | 1 <element>;

This example denotes a choice of either the set composed of a single 1-bit string – the bit string composed of a single bit of value 0 – or the set composed of the concatenation of a single 1-bit string – the bit string composed of a single bit of value 1 – and the bit string representing the element.

Message / Structured Element Items			
Name	**Type**	**Type Modifier**	**Comment**
element	CSN1_S1		Example item

Table 37: SAPE Table connected to the current example

In dependency of internal structure of <element> you have to choose the appropriate sub table.
If the element is a structured element you need another structured element definition table defining the internal element composition; see the table below:

<element> ::= <a> <b> <c>;

Structured Element Definition				
Name	**Type**	**Feature Flag**	**Group**	**Comment**
element	STRUCT			Structured Element
Structured Element Items				
Name	**Type**	**Type Modifier**	**Comment**	
a			Element a	
b			Element b	
c			Element c	

Table 38: SAPE Table belonging to the internal structure of the current example

If the element is a basic element you need another basic element definitions table that looks like the following example:

<element : bit(5)>

Basic Element Definitions					
Name	**Bit Len**	**Byte Len**	**Feature Flags**	**Group**	**Comment**
element	5				Basis element consisting 5 bits

Table 39: SAPE Table belonging exemplify a basic element

4.2.2.2 CSN1_S0

The CSN1_S0 coding type is an appropriate solution if an optional information element is described by the following example:

<something> ::= 1 | 0 <element>;

This example denotes a choice of either the set composed of a single 1-bit string – the bit string composed of a single bit of value 1 – or the set composed of the concatenation of a single 1-bit string – the bit string composed of a single bit of value 0 – and the bit string representing the element.

Message / Structured Element Items			
Name	**Type**	**Type Modifier**	**Comment**
element	CSN1_S0		Example item

Table 40: SAPE Table connected to the current example

In dependency of internal structure of <element> you have to choose the appropriate sub table (see the examples above).

4.2.2.3 CSN1_SHL

The special notations L and H are used to denote respectively the bit value corresponding to the padding spare bit for that position, and the other value.

The coding type CSN1_SHL supports elements comprising of a single bit valid flag and a value part. Only if the valid bit is equal to H the value part follows. Otherwise (valid bit is equal to L) the value part is absent.

Example:

```
<z>  ::= {   <a>
             { L | H <b : bit(3)> }
             < spare padding > ;
```

Structured Element Definition					
Name		**Type**	**Feature Flag**	**Group**	**Comment**
z		STRUCT			Structured Element
Structured Element Items					
Name	**Pattern**	**Bit Len**	**Type**	**CmdSequence**	**Comment**
a					Element a
b			CSN1_SHL		Element b
	00101011	8		0	Padding bits

Table 41: SAPE Table belonging to the internal structure of the current example

The coded value of the single bit flag following element <a> depends on its bit position.

Assumption 1:
The last bits of item <a> ends on octet **m** bit position 6 and element <b> is absent. Then bit 5 has to be set to 0 and bit4down to 1 will be filled with s_padding bits:

	8	7	6	5	4	3	2	1	
octet m	x	x	x	.	.	.	.	.	Rest of <a>
				0					Flag
	.	.	.		1	0	1	1	Padding Bits

Assumption 2:
The last bits of item <a> ends on octet **m** bit position 5 and element <b> is absent. Then bit 4 has to be set to 1 and bit 3 down to 1 will be filled with s_padding bits:

	8	7	6	5	4	3	2	1	
octet m	x	x	x	x.	.	.	.	.	Rest of <a>
					1				Flag
	.	.	.			0	1	1	Padding Bits

Assumption 3:
The last bits of item <a> ends on octet **m** bit position 6 and element <b> is present. Then bit 5 has to be set to 1 indicating presence of the element <b>. Bit 1 will be filled with s_padding bit:

	8	7	6	5	4	3	2	1	
octet m	x	x	x	.	.	.	.	.	Rest of <a>
				1	x	x	x		Flag and <b>

	.	.	.				1	Padding Bits

Assumption 4:
The last bits of item <a> ends on octet **m** bit position 5 and element <b> is present. Then bit 4 has to be set to 1 indicating presence of the element <b>. The octet is filled completely; therefore no spare padding bits are necessary.

	8	7	6	5	4	3	2	1	
octet m	x	x	x	x		.	.	.	Rest of <a>
					0	x	x	x	Flag and <b>

4.2.2.4 HL_FLAG

Like coding type CSN1_SHL this coding type is associated with the special notations L and H. They are used to denote respectively the bit value corresponding to the padding spare bit for that position, and the other value.

A HL_FLAG element consists of a single bit only. The decoded value will be 0 if the encoded value is L respectively 1 if the encoded value is H. This element enables support of a choice according to the following example:

<z> ::= { { L <a> } | { H <b> };
The SAPE message description should look like this:

Structured Element Definition				
Name	**Type**	**Feature Flag**	**Group**	**Comment**
z	STRUCT			Structured Element
Structured Element Items				
Name	**Type**	**Condition**	**Comment**	
flag	HL_FLAG			
a		{flag = 0}	Element a	
b		{flag = 1}	Element b	

Table 42: SAPE Table belonging to the structure using HL_FLAG coding type

Please keep in mind that the value 0 for flag is associated with the bit evaluation L. The value 1 refers to H.

4.2.2.5 CSN1_CONCAT

Truncated concatenation is a special sequence of components. In the technical specifications written by ETSI/3GPP you will find the following notation: The sequence of components encapsulated by the { } brackets is followed by the symbol '//'. This means that the concatenation is any of the concatenations starting with null and up to any number of components arranged in a definite sequence.

{ < a > < b > < c > } //

The above set is equivalent to:

{ < a > < b > < c > } or { < a > < b > } or { < a > } or null

The TI tool chain provides the coding type CSN1_CONCAT to support truncated concatenation. The sequence of components (truncated concatenation) must be handled as a structured information element associated with the coding type CSN1_CONCAT. This structured element comprises all components.

Structured Element Items			
Name	**Type**	**Type Modifier**	**Comment**
tmc_concat_comp	CSN1_CONCAT		Truncated Concatenation

Table 43: SAPE Table belonging to a truncated concatenation information element

Structured Element Definition				
Name	**Type**	**Feature Flag**	**Group**	**Comment**
tmc_concat_comp	STRUCT			Structured Element

Structured Element Items			
Name	**Type**	**Type Modifier**	**Comment**
a			Element a
b			Element b
c			Element c

Table 44: SAPE Table Belonging to the internal structure of a truncated concatenation information element

Any syntactically incorrect component shall truncate the sequence. The correctly received components are accepted and the truncated components are ignored. The TI Coder Decoder CCD recognizes any syntactical error and returns depending on the gravity of the situation either a warning or an error.

The truncated concatenation may include 'padding bits' at the end of a message. In that case, the resulting concatenation shall fit exactly with the received message length. Otherwise, it is a syntactical error, which may cause rejection of the complete message or part thereof. The TI tool chain detects syntactical errors, but the user has to decide how to deal with them.

The construction is useful, e.g. when a message ends with a sequence of optional components, where the transmitter may need to truncate tailing bits '0', indicating optional components not included in the message.

```
< Packet ZZZ message content > ::=
        ...
        {       { 0 | 1    < Optional component 1 > }
                { 0 | 1    < Optional component 2 > }
                ...
                { 0 | 1    < Optional component N > }
                < padding bits > } // ;
```

If the optional components from k to N are not needed in the message, the transmitter may use the full message length for the components up to optional component k – 1. The receiver accepts this message and assumes that the choice bits for optional components from k to N are all set to zero (i.e. these components are not present).

However, if the receiver detects a syntactical error within one optional component which is indicated as present in the message, that results in a truncated concatenation which does not fit with the received message length. In this case, the receiver shall not accept the message as being syntactically correct.

Note:
A truncated concatenation is a sequence of optional components. But in this case the meaning of the word <optional> differs from the traditional TI tool chain conventions.

So far some coding types (like tagged types, e.g. GSM3_TV) characterise optional elements inherently. If you describe a component in the message description by one of these coding types CCDGEN provides an appropriate element associated with a valid flag in the C structures while generating C header files. The value of the valid flag indicates the presence or absence of such an element.

Components concatenated with a CSN1 coding type cause these valid flags in the C header structure too. If you find a bit in the received message stream indicating optional values not included in the message (e. g. a CSN1_S1 element is represented by '0'), CCD will set the valid flag to zero. If this component belongs to a truncated concatenation the absence of the value does not truncate the sequence. The whole element represented by the flag is present; only the value is absent! The valid flag in the C structure is not an indication of an element's absence.

A truncated concatenation may comprise components associated with the coding types which do not characterise optional elements inherently. In this case CCDGEN provides appropriate elements without associated valid flags in the C structures; although it should be possible to truncate the sequence. The absence of an element associated with this kind of coding type truncates the sequence.

In case of truncated concatenations neither the absence of a valid flag nor a valid flag set to zero is a definite indication of an element's absence. Therefore you need an aid to recognize how many components could be decoded out of a received message stream.

It is recommended to use a leading element of coding type NO_CODE in the message description which is used to count the existing elements of the truncated concatenation. If this element is missing the decoding process will proceed but the CCD user is forced to evaluate the presence of optional components from k to N by himself. In case of decoding CCD writes the number of decoded elements belonging to the truncated concatenation to the NO_CODE element.

In case of encoding CCD always encodes all elements belonging to the truncated concatenation. If more bits are written than the component l_buf[11] of the message buffer suggested CCD generates a warning (error code ERR_BUFFER_OF). It is up to the user to analyse the consequences of this warning and to choose adequate procedures.

If the truncated concatenation not finishing the message description is followed by any other elements you will have another message element characterising the bit length of the truncated concatenation. In this case the structured element item must be associated with a type modifier indicating a bit string.

Example:

```
< Packet xxx message content > ::=
        <length : bit (6)>
        < bit (val(length) + 1)
        & { < trnc_concat_comp > ! { bit ** = <no string> }} >;

< trnc_concat_comp > ::=
        { < a > < b > < c > } // ;
```

Structured Element Definition				
Name	**Type**	**Feature Flag**	**Group**	**Comment**
Packet xxx message content	STRUCT			Structured Element
Structured Element Items				
Name	**Type**	**Type Modifier**	**Comment**	
length			Length identifier	
trnc_concat_comp	CSN1_CONCAT	[.length..MAX_LEN]	Truncated concatenation processed as bit string	

Table 45: SAPE Table belonging to the information element Packet xxx message content

Structured Element Definition				
Name	**Type**	**Feature Flag**	**Group**	**Comment**
trnc_concat_comp	STRUCT			Structured Element
Structured Element Items				
Name	**Type**	**Type Modifier**	**Comment**	
count	NO_CODE		only present in the C-structure to count the number of encoded elements	
a			Element a	
b			Element b	
c			Element c	

Table 46: SAPE Table belonging to the internal structure of the information element trnc_concat_comp

4.2.2.6 BREAK_COND

For decoding Neighbour Cell Parameter information elements belonging to GRR protocol the possibility is needed to stop decoding of a repeated structure by evaluating a bit field inside any repetition of the

[11] See ccd_api.doc : Data Type T_MSGBUF – Coded Message Data Type

structure and to continue decoding of following information elements. This feature is supported by the coding type BREAK_COND.

This special structure of information element is specified in [15.] 3GPP TS 44.060 part 11.2.21.

Simplified example using CSN.1 Notation:

```
<z>                    ::= { 1    < Repeat struct >
                               < x : bit (2) > } ** 0 ;

< Repeat struct >      ::= {    1    < a : bit(5)>
                               {    < number : { bit (4) – 0000 } >
                                    < b : bit (5) > * (val(number))
                                    < Repeat struct >          -- Repeated recursively
                               |    0000 }                     -- Break recursion (number == 0)
                             | 0 } ;                           -- End recursion (no more <a>)
```

Some examples for possible bit streams:

Example A)

1	
1 xxxxx a (5 bit) 0010 xxxxx b (5 bit) xxxxx b (5 bit) 0	< Repeat struct > number = 2 repeated Repeat struct
00	< x : bit (2) >
0	closing zero for repetition of {< Repeat struct > < x >}

Example B)

1	
1 xxxxx a (5 bit) 0000	< Repeat struct > number = 0
00	< x : bit (2) >
1	further repetition of {< Repeat struct > < x >}
1 ...	< Repeat struct > etc. finished by one of the methods a) or b) (see below)
01	< x : bit (2) >
0	closing zero for repetition of {< Repeat struct > < x >}

Example C)

1	
1 xxxxx a (5 bit) 0010 xxxxx b (5 bit) xxxxx b (5 bit) 1 xxxxx a (5 bit) 0011 xxxxx b (5 bit) xxxxx b (5 bit) xxxxx b (5 bit) 0	< Repeat struct > number = 2 repeated Repeat struct number = 3 repeated Repeat struct
00	< x : bit (2) >
0	closing zero for repetition of {< Repeat struct > < x >}

Example D)

1	
1 xxxxx a (5 bit) 0010 xxxxx b (5 bit) xxxxx b (5 bit) 1 xxxxx a (5 bit) 0001 xxxxx b (5 bit) 1 xxxxx a (5 bit) 0000	< Repeat struct > number = 2 repeated Repeat struct number = 1 repeated Repeat struct number = 0
00	< x : bit (2) >
0	closing zero for repetition of {< Repeat struct > < x >}

There are **two** mechanisms to finish a < Repeat struct >:

a) a 0 indicating the absence of a further repetition (in case the value of the element 'number' is different from zero)
b) the value of the element 'number' equal to zero (in this case the repetition **isn't** finished by a trailing 0 and there are no IE to decode following normally).

For case a) there is normally used the combination of coding type CSN1_S1 with an array, but for case b) this mechanism fails.

Usage of conditions to describe b) isn't sufficient because CCD expects the trailing 0 anyway.

The new coding type with the parameter n (in our example n=0) supports a behavior like "if a certain IE has the special value n then break (de)coding process of the current composition and finish the array.

This coding type is applicable to basic elements and forces CCD to compare the resulting value with n after decoding the requested number of bits.

SAPE representation:

Structured Element Definition				
Name	**Type**	**Feature Flag**	**Group**	**Comment**
z	STRUCT			Structured Element
Structured Element Items				
Name	**Type**	**Type Modifier**	**Comment**	
Repeat_struct	CSN1_S1	[0..MAX_ REPS]	BREAK_COND has impact to this CSN1_S1 type	
x				

Table 47: SAPE Table belonging to the information element z

Structured Element Definition				
Name	**Type**	**Feature Flag**	**Group**	**Comment**
Repeat_struct	STRUCT			Structured Element
Structured Element Items				
Name	**Type**	**Type Modifier**	**CmdSequence**	**Comment**
a				
number	BREAK_COND		0	Here: break = 0
b		[number..MAX_NUM]		

Table 48: SAPE Table belonging to the internal structure of the information element Repeat_struct

The corresponding type in the C-structure is available by chapter 6.

4.2.2.7 CSN1_CHOICE1 and CSN1_CHOICE2

These coding types should improve the en-/decoding process of messages with information elements which presence depends on leading flag values given by a single bit respectively two bits.

The coding type CSN1_CHOICE1 supports elements comprising of a single bit flag and two alternative value parts depending on this flag value. Only if the flag bit is equal to "0" the first value part follows and the second is absent. Otherwise (flag bit is equal to "1") the first value part is missing instead the second value part is present.

Example:

```
<msg_ex>   ::= {   <x>
                   {{ 0 <a>} | { 1 <b>}}
                   <z>}
```

The coding type CSN1_CHOICE1 enables the choice construction

```
<y>        ::= { 0 <a>} | {1 <b>};
```

to be handled as union.

In the structured element definition the user should associated the element <y> with the type UNION instead of the the type STRUCT. The SAPE message description should look like this:

Structured Element Definition				
Name	**Type**	**Feature Flag**	**Group**	**Comment**
msg_ex	STRUCT			Structured Element
Structured Element Items				
Name	**Type**	**Condition**	**Comment**	
x			Element x	
y	CSN1_CHOICE1		Element y	
z			Element z	

Table 49: SAPE Table belonging to the structure comprising an element of CSN1_CHOICE1 coding type

Structured Element Definition					
Name	**Type**	**Feature Flag**	**Group**	**Comment**	
y	UNION			Structured Element	
Structured Element Items					
Name	**Pattern**	**Bit Len**	**Type**	**Union Tag**	**Comment**
a				choiceA_1	Element a if flag = 0
b				choiceA_2	Element b if flag = 1

Table 50: SAPE Table belonging to the internal structure of the CSN1_CHOICE1 element

The coding type CSN1_CHOICE2 is very similiar, but instead of one bit flag indicating the choice element there are two bits:

Example:

```
<msgex>    ::= {   <x>
                   {{ 00 <a>} | { 01 <b>} | { 10 <c>} | { 11 <d>} }
                   <z>}
```

The coding type CSN1_CHOICE2 enables the choice construction

```
<y>        ::= { 00 <a>} | { 01 <b>} | { 10 <c>} | { 11 <d>};
```

to be handled as union.

In the structured element definition the user should associated the element <y> with the type UNION instead of the the type STRUCT. The SAPE message description should look like this:

Structured Element Definition				
Name	**Type**	**Feature Flag**	**Group**	**Comment**
msg_ex	STRUCT			Structured Element
Structured Element Items				
Name	**Type**	**Condition**	**Comment**	
x			Element x	
y	CSN1_CHOICE2		Element y	
z			Element z	

Table 51: SAPE Table belonging to the structure comprising an element of CSN1_CHOICE2 coding type

Structured Element Definition					
Name	**Type**	**Feature Flag**	**Group**	**Comment**	
y	UNION			Structured Element	
Structured Element Items					
Name	**Pattern**	**Bit Len**	**Type**	**Union Tag**	**Comment**
a				choiceB_1	Element a if flag = 00
b				choiceB_2	Element b if flag = 01
c				choiceB_3	Element a if flag = 10
d				choiceB_4	Element b if flag = 11

Table 52: SAPE Table belonging to the internal structure of the CSN1_CHOICE2 element

If there are not all alternatives given in the specification the user is expected to list the unused flag combinations concatenated with NO_CODE element anyway! Otherwise CCDGEN will complain.

4.2.2.8 CSN1_S1_OPT

The CSN1_S1_OPT coding type is an appropriate solution if an optional information element is described by the following example:
<something> ::= null | 0 | 1 <element>;

This coding type is very similar to the coding type CSN1_S1, which doesn't support the possibility of element lack. CSN1_S1_OPT is the appropriate solution if the specification demands a construction with a composition of coding type CSN1_CONCAT and an encapsulated CSN1_S1 element. This type increases the recursions level and therefor the runtime performance would be decreased.

This example denotes a choice of either the set composed of a single 1-bit string – the bit string composed of a single bit of value 0 – or the set composed of the concatenation of a single 1-bit string – the bit string composed of a single bit of value 1 – and the bit string representing the element. Besides the absence of the whole element is allowed.

Message / Structured Element Items			
Name	**Type**	**Type Modifier**	**Comment**
element	CSN1_S1_OPT		Example item

Table 53: SAPE Table belonging to an element of coding type CSN1_S1_OPT

In dependency of internal structure of <element> you have to choose the appropriate sub table (see the examples for CSN1_S1).

4.2.2.9 CSN1_S0_OPT

The CSN1_S0_OPT coding type is an appropriate solution if an optional information element is described by the following example:

<something> ::= null | 1 | 0 <element>;

This coding type is very similar to the coding type CSN1_S0, which doesn't support the possibility of element lack. CSN1_S0_OPT is the appropriate solution if the specification demands a construction with a composition of coding type CSN1_CONCAT and an encapsulated CSN1_S0 element. This type increases the recursions level and therefor the runtime performance would be decreased.

This example denotes a choice of either the set composed of a single 1-bit string – the bit string composed of a single bit of value 1 – or the set composed of the concatenation of a single 1-bit string – the bit string composed of a single bit of value 0 – and the bit string representing the element. Besides the absence of the whole element is allowed.

Message / Structured Element Items			
Name	**Type**	**Type Modifier**	**Comment**
element	CSN1_S0_OPT		Example item

Table 54: SAPE Table belonging to an element of coding type CSN1_S0_OPT

In dependency of internal structure of <element> you have to choose the appropriate sub table (see the examples for CSN1_S1).

4.2.2.10 CSN1_SHL_OPT

The special notations L and H are used to denote respectively the bit value corresponding to the padding spare bit for that position, and the other value.

The coding type CSN1_SHL_OPT supports elements comprising of a single bit valid flag and a value part. Only if the valid bit is equal to H the value part follows. Otherwise (valid bit is equal to L) the value part is absent. Besides the absence of the whole element is allowed.

This coding type is very similar to the coding type CSN1_SHL, which doesn't support the possibility of element lack. CSN1_SHL_OPT is the appropriate solution if the specification demands a construction with a composition of coding type CSN1_CONCAT and an encapsulated CSN1_SHL element. This type increases the recursions level and therefor the runtime performance would be decreased.

Other details fit to coding type CSN1_SHL (see part 4.2.2.3).

4.2.3 Special Coding Types

4.2.3.1 S_PADDING

An issue appearing in some protocols, for instance the GSM radio interface protocols, is that in some cases the useful part of a message is smaller than some fixed length imposed by underlying layers. Padding bits are then necessary to fill the message up to the desired length. The padding bits may be the 'null' string.

An issue specific to [12.] 3GPP TS 24.008 is that the padding uses a particular sequence of bit, of fixed position, i.e., the value of a padding bit depends on its position relative to the start of the message. The padding sequence must then be considered as protocol specific. In the case of 3GPP TS 24.008 the padding sequence used for 'spare padding' is a repetition of octet '00101011', starting on an octet boundary.

The TI tool chain offers the coding type S_PADDING to support different spare padding patterns and to enable the appropriate handling of the insignificant but necessary fill bits. When CCDGEN generates the C header files comprising C structures this coding type advises CCDGEN to disregard the associated message/structured element item. But during the transformation of message description files (*.mdf) to ccddata tables (*.cdg) CCDGEN creates the required table entries for these elements and their patterns.

Example for the usage in SAPE:

Structured Element Items						
Name	**Pattern**	**Bit Len**	**Type**	**Type Modifier**	**CmdSequence**	**Comment**
	00101011	8	S_PADDING		22	Spare padding according to 3GPP TS 24.008

Table 55: SAPE Table representing an Spare Padding IE according to [12.] 3GPP TS 24.008

The definition of a Spare element must be associated with an appropriate Command Sequence Element. This PadCmdSeq expression determines the maximum length of spare padding bits. The used number n means if a Structured Message Element or a Message consists of less than n bytes the remaining part shall be filled up with the bit pattern given by the Pattern element. Padding bytes exclude the presence of message extension. The usage of number n = 0 denotes a special case: If the message doesn't end on octet boundary the remaining part up to the next octet boundary shall be filled up with the rest of the bit pattern given by the Pattern element on the appropriate bit positions.

In case of decoding CCD does not evaluate the encoded bits, since their content is irrelevant.

In case of encoding of an element associated with the coding type S_PADDING until now CCD supports only two padding values: 0x00 and 0x2B. If the Command Sequence Element is n = 0 CCD fills the current octet up to its boundary with padding bits according to the given pattern. If the Command Sequence Element is n > 0 padding is done as far as the message bit stream pointer reaches the position n*8. Using unsupported padding pattern causes a CCD error indication and breaks the encoding process.

4.2.3.2 S_PADDING_0

Very often the padding bits starts with bit '0', followed by 'spare padding' to enable support of future message extensions.

```
< padding bits > ::= { null | 0 < spare padding > ! < Ignore : 1 bit** = < no string > > } ;
```

The bit '0' in the first bit position of the 'padding bits' may be altered into a bit '1' in future versions of the present document, in order to indicate an extension of the message content. When a message is received with bit '1' in this position, a receiver implemented according to the current version of the present document shall ignore the remaining part of the message.

The presence of the extension of the message content is indicated by bit '1'. The transmitter shall send a bit '1' in this position if any content is defined for the remaining part of the message. If a bit '0' is received in this position by a receiver in the new version, it shall ignore the remaining part of the message.

Besides the features described above (see part S_PADDING) persist.

4.2.3.3 Frequency List Information

The purpose of the *Frequency List* information element is to provide the list of the absolute radio frequency channel numbers used in a frequency hopping sequence.
There are several formats for the *Frequency List* information element, distinguished by the "format indicator" subfield. Some formats are frequency bit maps; the others use a special encoding scheme.
The Generic Tool Chain provides one coding types (FREQ_LIST) which support all different formats. There are two other coding types (FDD_CI and TDD_CI) each supporting only a special subset of formats. All three types are closely related.

These coding types support decoding of RR Frequency List, fdd_cell_information and tdd_cell_information by CCD. You will find a detailed description of the algorithm to encode/decode frequency list information elements Annex J of the specification [14.] 3GPP TS 44.018.

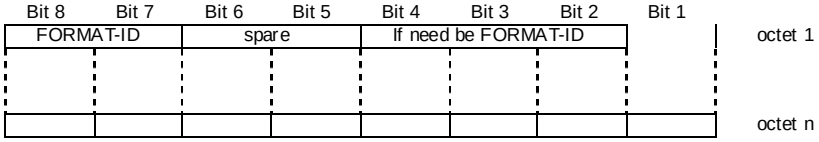
Some information elements encode frequency lists with a special method. The main specification specifies the meaning of the fields and hence the way to decode them, but the corresponding encoding algorithm is difficult to infer from the decoding algorithm. The specification [14.] 3GPP TS 44.018 is intended as an aid for implementers of the encoding algorithm.

So far encoding functions are not supported, because messages using this information are sent from core network to mobile stations.

FREQ_LIST – RR Frequency List

General description

Frequency List information element, general format



The different formats are distinguished by the FORMAT-ID field. The possible values are the following:

Bit 8	Bit 7	Bit 4	Bit 3	Bit 2	format notation
0	0	X	X	X	bit map 0
1	0	0	X	X	1024 range
1	0	1	0	0	512 range
1	0	1	0	1	256 range
1	0	1	1	0	128 range
1	0	1	1	1	variable bit map

All other combinations are reserved for future use.

The significance of the remaining bits depends on the FORMAT-ID.

Example for the usage in SAPE:

Message Items						
Name	**Pattern**	**Bit Len**	**Type**	**Type Modifier**	**CmdSequence**	**Comment**
arfcn_list			FREQ_LIST			arfcn_list

Table 56: SAPE Table representing an information element of coding type FREQ_LIST

The element *arfcn_list* should be a basic element according to the following basic element definitions table

Basic Element Definitions					
Name	**Bit Len**	**Byte Len**	**Feature Flags**	**Group**	**Comment**
arfcn_list	1024				This list contains absolout radio frequency channel numbers to be decoded as FREQ_LIST type.

Table 57: SAPE Table belonging exemplify the associated basic element

CCDGEN accepts elements of coding type FREQ_LIST and generates appropriate ccddata table entries and header file structures. This coding type is not optional and does not require additional bits (addbits = 0). The corresponding type definition in the C-structure will be an array of unsigned char (see chapter 6).

CCD provides an appropriate function for decoding. This tool recognizes the FORMAT-ID field and chooses the necessary algorithm to convert a frequency bit map to an array of unsigned short variables. Present frequency values are transferred to the array. So the array will contain a sorted list of frequency values marked as present in the bit map. CCD creates a sorted frequency hopping list for one of the following information elements according to [12.] 3GPP TS 24.008:

cell channel description
frequency list

frequency short list
neighbour cell description

FDD_CI - fdd_cell_information

For decoding of the structure <Repeated UTRAN FDD Neighbour Cells> belonging to RRC protocol a convenient possibility is needed to allow computation of a set of 10-bit-long FDD_CELL_INFORMATION parameters.

This feature is supported by the coding type FDD_CI re-using the Range 1024 format compression algorithm, see [14.] 3GPP TS 44.018 Annex J: 'Algorithm to encode frequency list information'.

This special structure of information element is specified in [14.] 3GPP TS 44.018 part 9.1.54.

CSN.1 Notation:

< Repeated UTRAN FDD Neighbour Cells struct > ::=
  0 < **FDD-ARFCN** : bit (14) >
    < **FDD_Indic0** : bit >
    < **NR_OF_FDD_CELLS** : bit (5) >
    < **FDD_CELL_INFORMATION Field** : bit(p(NR_OF_FDD_CELLS)) > ;
                                        -- p(x) defined in table below.

The total number of bits p of this field depends on the value of the parameter NR_OF_FDD_CELLS = n, as follows:

n	p	n	p	n	p	n	p
0	0	5	44	10	81	15	116
1	10	6	52	11	88	16	122
2	19	7	60	12	95	17-31	0
3	28	8	67	13	102		
4	36	9	74	14	109		

If n=0 and FDD_Indic0=0, this indicates the 3G Neighbour Cell list index for report on RSSI, see 3GPP TS 45.008.

If n is equal or greater than 17, this shall not be considered as an error; the corresponding index in the 3G Neighbour Cell list shall be incremented by one.

For each (10-bit-long) decoded Parameter, bits 1-9 are the Scrambling Code and bit 10 is the corresponding Diversity Parameter.

Example for the usage in SAPE:

Structured Element Definition				
Name	**Type**	**Feature Flag**	**Group**	**Comment**
fdd_ci_cmp	STRUCT			Structured Element containing fdd_ci arrays
Structured Element Items				
Name	**Type**		**Type Modifier**	**Comment**
fdd_arfcn				UARFCN
fdd_indic_0				information 0 indicator
fdd_cell_information	FDD_CI		[0..17]	

Table 58: SAPE Table representing with an information element of coding type FDD_CI

Basic Message Elements					
Name	**Bit Len**	**Byte Len**	**Feature Flags**	**Group**	**Comment**
fdd_arfcn	14				UARFCN
fdd_indic_0	1				information 0 indicator
fdd_cell_information	10				scrambling code and diversity information

Table 59: SAPE Table belonging exemplify the associated basic elements

The IE FDD_CELL_PARAMETERS is associated with the coding type FDD_CI. CCDGEN accepts elements of coding type FDD_CI and generates appropriate ccddata table entries and header file structures. The coding type is not optional and does not require additional bits (addbits = 0). The corresponding type in the C-structure will be an array of unsigned short values. The array size is given by the MAX value in message description document. The user has to define this maximum value for the TypeModifier controlling the array length according to his knowledge of specification or test experience.

CCD provides an appropriate function for decoding. First CCD reads 5 bits from the received data stream defining the number of FDD_CELL_INFORMATION parameters (**n**). This number enables CCD to decide how many values of different word length are present in the encoded message. FDD_CELL_PARAMETERS or "Scrambling Codes and Diversity Field" is a bit filed of length **p** (Number_of_Scrambling_Codes_and_Diversity), whereas the function **p**(x) is defined by the table above with **n** = FDD_CELL_INFORMATION parameters.

Decoding of the FDD_CELL_PARAMETERS Field reuses the RANGE 1024 format of frequency list information. The IE FDD_CELL_PARAMETERS is preceded by FDD_Indec0 (1 bit) indicating if the parameter value '0000000000' is a member of the set. This bit FDD_Indic0 is equivalent to the bit F0 bit in the frequency list information element (see [14.] 3GPP TS 44.018 part 10.5.2.13.3). The set of possible values is almost the same for frequency list and fdd_cell_information.

The corresponding type in the C-structure will be an array of unsigned short values (see chapter 6).

The message is sent from core network to mobile stations. These mobiles supporting enhanced measurements have to understand it. The space this IE takes in the C-structure depends on a counter for the number of decoded parameters and the array of them.

TDD_CI - tdd_cell_information

This coding type is very similar to FDD_CI. For decoding of the structure < Repeated UTRAN TDD Neighbour Cells > belonging to RRC protocol a convenient possibility is needed to allow computation of a set of 9-bit-long TDD_CELL_INFORMATION parameters.

This feature is supported by the coding type TDD_CI re-using the *Range 512 format* compression algorithm, see [14.] 3GPP TS 44.018 Annex J: 'Algorithm to encode frequency list information'.

This special structure of information element is specified in [14.] 3GPP TS 44.018 part 9.1.54.

CSN.1 Notation:

< Repeated UTRAN TDD Neighbour Cells struct > ::=
  0 < **TDD-ARFCN** : bit (14) >
    < **TDD_Indic0** : bit >
    < **NR_OF_TDD_CELLS** : bit (5) >
    < **TDD_CELL_INFORMATION Field** : bit(q(NR_OF_TDD_CELLS)) > ;
                                        -- q(x) defined in table below.

The total number of bits p of this field depends on the value of the parameter NR_OF_TDD_CELLS = m, as follows:

m	q	m	q	m	q	m	q	m	q
0	0	5	39	10	71	15	101	20	126
1	9	6	46	11	77	16	106	21-31	0
2	17	7	53	12	83	17	111		
3	25	8	59	13	89	18	116		
4	32	9	65	14	95	19	121		

If m=0 and TDD_Indic0=0 or m is equal or greater than 21, this shall not be considered as an error; the corresponding index in the 3G Neighbour Cell list shall be incremented by one.

For each (9-bit-long) decoded Parameter, bits 1-7 are the Cell Parameter, bit 8 is the Sync Case and bit 9 is the Diversity bit.

Example for the usage in SAPE:

Structured Element Definition				
Name	**Type**	**Feature Flag**	**Group**	**Comment**
tdd_ci_cmp	STRUCT			Structured Element containing tdd_ci arrays
Structured Element Items				
Name	**Type**		**Type Modifier**	**Comment**
tdd_arfcn				UARFCN
tdd_indic_0				information 0 indicator
tdd_cell_information	TDD_CI		[0..21]	

Table 60: SAPE Table representing with an information element of coding type FREQ_LIST

Basic Message Elements					
Name	**Bit Len**	**Byte Len**	**Feature Flags**	**Group**	**Comment**
tdd_arfcn	14				UARFCN
fdd_indic_0	1				information 0 indicator
tdd_cell_information	10				scrambling code and diversity information

Table 61: SAPE Table belonging exemplify the associated basic elements

The IE TDD_CELL_PARAMETERS is associated with the coding type TDD_CI. CCDGEN accepts elements of coding type TDD_CI and generates appropriate ccddata table entries and header file structures. The coding type is not optional and does not require additional bits (addbits = 0). The corresponding type in the C-structure will be an array of unsigned short values. The array size is given by the MAX value in message description document. The user has to define this maximum value for the TypeModifier controlling the array length according to his knowledge of specification or test experience.

CCD provides an appropriate function for decoding. First CCD reads 5 bits from the received data stream defining the number of TDD_CELL_INFORMATION parameters (**m**). This number enables CCD to decide how many values of different word length are present in the encoded message. TDD_CELL_PARAMETERS or "Scrambling Codes and Diversity Field" is a bit filed of length **q** (Number_of_Scrambling_Codes_and_Diversity), whereas the function **q**(x) is defined by the table above with **m** = TDD_CELL_INFORMATION parameters.

Decoding of the TDD_CELL_PARAMETERS Field reuses the RANGE 512 format of frequency list information. The IE TDD_CELL_PARAMETERS is preceded by TDD_Indec0 (1 bit) indicating if the parameter value '0000000000' is a member of the set. This bit TDD_Indic0 is equivalent to the bit F0 bit in the frequency list information element (see [14.] 3GPP TS 44.018 part 10.5.2.13.3). The set of possible values is almost the same for frequency list and tdd_cell_information.

The corresponding type in the C-structure will be an array of unsigned short values (see chapter 6).

The message is sent from core network to mobile stations. These mobiles supporting enhanced measurements have to understand it. The space this IE takes in the C-structure depends on a counter for the number of decoded parameters and the array of them.

4.2.3.4 NO_CODE

An IE of this coding type supports two functionalities: either reading of a value from the stack and writing it into the C structure or processing the error branch in case of a message escape error label. An IE of this type does not occur in the air message. Nevertheless in case of encoding the variable in the C structure must be written by the caller entity, because encoding of some other message elements could need this information.

EXAMPLE (reading stack value):
The first usage of this type is the IE "tlv_len" in a Multi Rate Configuration. In this case "tlv_len" is to be used for evaluating conditions which decide on the content of Multi Rate Configuration IE.

EXMAPLE (message escape):
A part of a message, which depends on a certain protocol status, is marked by the 'Message escape'

error label. It is preceded by an amount of bits given by the specification. Some of these bit combinations are concatenated with a well-defined message structure. All the rest of combinations are expected to provide an escape.

4.2.4 Some tricky descriptions for particular message elements

4.2.4.1 Error Labels

This part describes the implementation of the following rules which are extracted from section 11.1 in [15.] TS 44.060.

- Unknown message type
- Syntactically incorrect message with different Error Labels
- Syntactic error in truncated concatenation

Generic Error Labels

Unknown Message Type

If a message with an unknown message type is recognized during decoding process CCD returns an error indication and breaks decoding.

⇒ Therefore no handling is required.

Certain Message Part Errors

'Distribution part error', 'Address information part error' and 'Non-distribution part error' are other generic error labels defined for syntactical errors. These error labels allow ignoring a part of the message that is syntactically incorrect. Once an error is detected, the error branch is called. The error branch is followed by an unspecified bit string that expands to the end of the message. The corresponding data is ignored (i.e. CCD breaks decoding process).

As yet CCD provides information on which IE an error is reported. With Error Labels CCD is be able to recognize the message part (Distribution, Address Information or NonDistribution Part) a reported error belongs to; besides CCD provides information about this erroneous part.

How to use this feature?

- In SAPE we use the **TypeModifier** Column to mark the beginning of a certain part belonging to a message which is classified either as distribution or non-distribution message.
- We concatenate the first element of each part with one of the following keywords:
  EL_AIP Address Information Part
  EL_DP Distribution Part
  EL_NDP Non-Distribution Part
- The converter 'xml2mdf' tunnels this information to mdf-File.
- CCDGEN processes this information and writes for each first IE of a special part a well-defined entry to the table cald.cdg.
  '**Z**' -> **EL_AIP** Address Information Part
  '**D**' -> **EL_DP** Distribution Part
  '**N**' -> **EL_NDP** Non-Distribution Part
- CCD reads the information from the calc-table. If CCD detects an error during the decoding process the Error Code will be changed to the appropriate error label and the decoding process will be broken.

Example

The *general format* of a non-distribution message, including these error labels is:

```
< Non-distribution message > ::=
    < MESSAGE_TYPE : 0 bit (5) >
    {       < Distribution contents >
            {       < Address information >
                    {       < Non-distribution contents >
                            < padding bits >
                             ! < Non-distribution part error : bit (*) = < no string > > }
                     ! < Address information part error : bit (*) = < no string > > }
             ! < Distribution part error : bit (*) = < no string > > }
     ! < Unknown message type : bit (6) = < no string > < Default downlink message content > > ;
```

SAPE representation:

Message Items				
Name	**Pattern**	**BitLen**	**TypeModifier**	**Comment**
MESSAGE_TYPE		6		
distribution_contents			~EL_DP	
address_information			~EL_AIP	
non_distribution_contents			~EL_NDP	
	00101101	8	S_PADDING	

Table 62: SAPE Table with Different Error Labels

Additional Error Labels

Besides the generic error labels there are the error labels 'Ignore' and 'Message Escape' which are allowed to be present in all message parts (Distribution part, Address information part and Non-distribution part). These kinds of error labels require a special error detection mechanism.

Message Escape Error Label

The 'Message escape' error label is used to provide an escape for, e.g., a future modification of the message syntax. Therefore the part of a message, which depends on a certain protocol status, is preceded by an amount of bits given by the specification. Some of these bit combinations are concatenated with a well-defined message structure. All the rest of combinations are expected to provide an escape.

How to use this feature?

- In SAPE we insert a flag element, which is used to keep the bits determining the kind of following message structure. Depending on the value of this flag element one of the following different message items is chosen. The unreserved flag values are associated with a message item of coding type NO_CODE. The concatenation of a certain flag value with a special message item is done by a condition written in the **Condition** Column (choice by flag value).
  We use the **TypeModifier** Column to mark the NO_CODE element with the Message Escape Error Label (**EL_MSG_ESC**).
- The converter 'xml2mdf' tunnels this information to mdf-File.
- CCDGEN processes this information and writes a well-defined entry to the table cald.cdg.
  '**M**' -> **EL_MSG_ESC**

  If CCD detects the presence of a NO_CODE element, CCD reads the information from the calc-table. The value in the calc-table determines whether the decoding process performs an error handling function or not. In case of a present Error Label

'Message Escape' the Error Code will be ERR_EL_MSG_ESC and the decoding process will be broken.

Example

< Packet YYY message content > ::= -- *Protocol version 1*
< **FIELD_1** : bit (3) >
{ 0 < **FIELD_2** : bit (16) >
< **FIELD_3** : bit (16) >
< padding bits >
! < Message escape : 1 bit (*) = <no string> > } ;

SAPE representation:

Structures Element Definitions				
Name	**Type**	**FeatureFlags**	**Group**	**Comment**
PACKET_YYY	STRUCT			Structured Element

Structures Elements Items						
Name	**Pattern**	**BitLen**	**Type**	**TypeModifier**	**Condition**	**Comment**
FIELD_1		3				
flag		1				
COMP1					{flag = 0}	
dummy			NO_CODE	EL_MSG_ESC	{flag # 0}	

Structures Element Definitions				
Name	**Type**	**FeatureFlags**	**Group**	**Comment**
COMP_1	STRUCT			Structured Element

Structures Element Items						
Name	**Pattern**	**BitLen**	**Type**	**TypeModifier**	**CmdSequence**	**Comment**
FIELD_2		16				
FIELD_3		16				
	00101101	8	S_PADDING		MAX_PADD	

Table 63: SAPE Table - Error Label Message Escape

Ignore Error Label

This error label allows ignoring a part of the message that is syntactically incorrect. Once the error is detected, the error branch 'Ignore' is called followed by an unspecified bit string. When this error label is used with an indefinite length (bit (*) = < no string >), the unspecified bit string expands to the end of the message and the corresponding data is ignored.

When this error label is used with a definite length (bit (n) = < no string >), the unspecified bit string contains a defined number of bits. The corresponding data is ignored. This feature will be an enhancement which isn't supported in the first implementation of error labels.

How to use this feature?

- In SAPE we use the **TypeModifier** Column to mark the beginning of a certain structure, which should be ignored in case of error detection.
- We concatenate the first element of this structure with the following keyword: **EL_IGNORE**. Later implementations will support this keyword in conjunction with the parameter *numBits*: **EL_IGNORE(numBits)**. This parameter will advise CCDGEN to add an additional row in the calc-table. CCD will process this constant value to skip a defined number of bits in case of error detection in conjunction with error labels.
- The converter 'xml2mdf' tunnels this information to mdf-File.
- CCDGEN processes this information and writes for first IE of marked structure a well-defined entry to the table cald.cdg. In future versions the parameter *numBits* will advise CCDGEN to add an additional row in the calc-table. CCD will process this constant value

to skip a defined number of bits in case of error detection in conjunction with error labels.
'**I**' -> **EL_IGNORE**

CCD reads the information from the calc-table. If CCD detects an error during the de-coding process the Error Code will be changed to the appropriate error label and the decoding process will be broken.

Example

< Packet XXX message content > ::=
< **FIELD_1** : bit (3) >
< **FIELD_2** : bit (16) >
< **FIELD_3** : bit (5) >
< padding bits >
! < *Ignore : bit (*) = < no string > >*

In the case of a complete message, the contents of the received syntactically incorrect message can be ignored.

SAPE representation:

Structures Element Definitions						
Name	**Type**		**FeatureFlags**	**Group**	**Comment**	
PACKET_YYY	STRUCT				Structured Element	
Structures Elements Items						
Name	**Pattern**	**BitLen**	**Type**	**TypeModifier**	**CmdSequence**	**Comment**
FIELD_1		3		~EL_IGNORE		
FIELD_2		16				
FIELD_3		5				
	00101101	8	S_PADDING		MAX_PADD	

Table 64: SAPE Table - Error Label Ignore

Conclusion

- Different types of error label shall be inserted in the TypeModifier-Column. The following error labels are supported by CCD and CCDGEN:

  EL_DPE < Distribution part error : bit (*) = < no string > >

  EL_AIPE < Address information part error : bit (*) = < no string > >

  EL_MSG_ESC < Message escape : 1 bit (*) = < no string > >

  EL_NDPE < Non-distribution part error : bit (*) = < no string > >

  EL_IGNORE < Ignore : bit (*) = < no string > >

  EL_I_TO_IE_END < Ignore : bit (n) = < no string > >
- Except EL_MSG_ESC the statement about error label must start with the character "~".
- The assumption of CCD is to "ignore" the remaining part of the message for most of the error labels and ignore the rest of the IE in case of EL_IGNORE.
- The entity calling CCD shall invalidate the IE at which error occurred.

4.2.4.2 How to express non-standard length information

Some protocols specify elements with variable length which can not be described standard LV-coding types – either the length information is not coded as 7 respectively 8 bit value or the length information and value are not submultiples of one unit. The length information and the value do not appear back-to-back.

Example using CSN.1 Notation:

```
<z>   ::= {    <length : bit(5)>
               <b : bit(val(length))> };
```

An alternative notation is:

```
<z>   ::= {    <length : bit(5)>
               <bit(val(length))>
               & { <b> ! {bit** = <no string>}} > };
```

The last notation rewords that both parts together - bit(val(length)) and <b> - must comprise so many bits as given by the value of the length element.

The SAPE message description should look like the following table:

Structured Element Definition					
Name		**Type**	**Feature Flag**	**Group**	**Comment**
z		STRUCT			Structured Element
Structured Element Items					
Name	**Pattern**	**Bit Len**	**Type**	**TypeModifier**	**Comment**
length					Length (Basic Element)
b				[.length .. MAX_LEN]	Element b

Table 65: SAPE Table belonging to an element of variable length

In this case CCD handles the information element <b> as a bit string independent of the internal structure of <b>. The element <b> may be a basic or a structured element. The length of this bit string must always depend on another basic IE declared before, just as in the example above.

A structured IE given as bit string will not be handled as an array type. In the header file generated by CCDGEN you will not find any counter for it. The bit size of the structure can only be calculated during the decoding real time processing. The calculation depends on the expressions either in the *Control* column of AIMs in MS-Word format or in the *TypeModifier* column of AIMs in SAPE format ([.length] in example above).

So far in case of encoding of such a bit string it is not possible to determine the length value depending on its nested encoded elements. The value of the corresponding length variable must be calculated and set by the CCD user.

5 CCDDATA

CCDDATA is a library that represents the CCD tables and concerning constants generated by CCDGEN. It must always be rebuilt if the interfaces defined in the SAP and MSG catalogues have changed.

The following sections give a detailed description of the CCDGEN output files with the name extension *.cdg. Most of the files will contribute to the CCD coding/decoding tables. The entries in the files are C-expressions. Each line is actually a member of a structure field.

- Table-oriented description of structures and values
- independent of h-files
- Table information
  - ⇒ Messages, Primitives, Structures
    - ♦ name, size (bit and byte oriented), number of elements
  - ⇒ Elements
    - ♦ type, coding type, repetition, offset in structure, optional, id, …
    - ♦ name, size (bit and byte oriented), values
- Names are usually omitted on targets
- Library Ccdedit provides a more readable interface to tables
  - ⇒ useful for test- and trace-tools

5.1 ccdmtab.cdg and ccdptab.cdg

These files contain C-expressions which include the relevant ***.cdg** files in order to define and initializes the so-called CCD tables. The tables used by CCD only are **mvar**, **spare**, **calc**, **mcomp**, **melem** and **mmtx**. They contain the rules needed by CCD for coding and decoding the specified messages. The CCD files used by CCDTAP (CCDEDIT) are **pvar**, **pcomp**, **pelem** and **pmtx**. A typical entry of **ccdmtab.cdg** is shown below:

```
const T_CCD_VarTabEntry   mvar  [] =
{
#include "mvar.cdg"
};
```

5.2 mstr.cdg

Although CCD does not use this table at the moment we do give a short description about it because of its simple structure. This is a simple formatted file containing either comments or long names related to the IEs. The entries can be used for example by the test systems while tracing the activities or errors. Some entries occur repeatedly, e.g. the words „Message Type“ and „reserved“. There are no entries for message names.

Generally the entries have the same order as they have in the word documents. In the example below, the long name of the first IE "Access identity" is followed by the two value entries, according to the chapter „basic elements“ of CC.doc.

```
/*  0*/ "",
/*  1*/ "Access identity",
/*  2*/ "octet identifier",
/*  3*/ "reserved",
```

5.3 mconst.cdg

A big part of these files contains definitions of C-macros for IDs, bit lengths etc for all entities (or SAPs). The other part is dedicated to either the C-macros for coding type or the constant values calculated by CCDGEN. The constants defined in **mconst.cdg** are important for coding/decoding and are included by CCD. However the constants in **pconst.cdg** relate to the SAPs and are only included by the tool CCDEDIT, which includes also the **mconst.cdg**. Below a few examples from **mconst.cdg** are given:

```
#define DATA_REQ                        0x1
#define BSIZE_DATA_REQ                  0x830       /* max bitlength of coded msg     */
...
#define CCDTYPE_GSM1_V                  0x1
#define CCDTYPE_GSM1_TV                 0x2
...
#define NUM_OF_ENTITIES                 0x6         /* number of entitys that uses CCD */
#define MAX_MESSAGE_ID                  0x7e        /* maximum of all message types   */
...
```

5.4 mvar.cdg and pvar.cdg

The table **mvar** contains specifying parameters for variables. CCD needs these parameters to decide for the bit or byte size and value of the variables. The format of the entries is given on the top of the list.

```
/* idx name lnameRef bSize cSize cType numValDefs valDefRef */
/*  0*/ { "msg_type"                ,    1,    8,   1, 'B',   0, 65535 },
/*  1*/ { "msg_id"                  ,    2,    8,   1, 'B',   0, 65535 },
/*  2*/ { "rel_mode"                ,    3,    8,   1, 'B',   2,     0 },
```

The parameter name has been called short **name** in the previous chapters. The parameter **lnameRef** gives the index referring to the entry in **mstr.cdg**. Bit lenght or **bSize** must be defined by the GSM protocol while **cSize** gives the byte size for the variable used in the CCD implementation. The possible entries for the member **cType** are B (boolean), S (Short) and X. The member **numValDefs** is the number of possible values for the variable. **valDefRef** is an index referring to the first entry for this IE in the table mval. We say the first entry because there must be **numValDefs** entries for an IE in mval. If there is no value supposed for an IE **numValDefs** will be 0 and **valDefRef** will be 65535.

Again **pvar** is only used by CCDEDIT but mvar is needed by CCDEDIT and CCD.

5.5 mval.cdg

Although CCD does not use this table at the moment we do give a short description about it because of its simple structure. The table **mval** contains specifying parameters for variable values. CCD needs these parameters to read a single value, a value range. The format of the entries is given on the top of the list.

```
/* idx valStrRef isDef startVal endVal */
/*  0*/ {    2, 0, 0x00000000, 0x00000000},
/*  1*/ {    3, 1, 0x00000000, 0x00000000},
```

The member **valStrRef** is the index referring to the comment in mstr.cdg about the specific value. The member **isDef** is a flag set to 1 whenever the given value is a default one for the corresponding variable.

The first (**startVal**) and last (**endVal**) value numbers determine the Value ranges. For single values the **startVal** is equal to **endVal**.

5.6 spare.cdg

This table contains value and bit length information for spare bits. Spare bits are often a series of zeros, which help to fill up an octet. The format of the entries is given on the top of the list.

```
/* idx spareValue bSize */
/*  0*/ { 0x00000000,   3},
/*  1*/ { 0x00000000,   7},
/*  2*/ { 0x00000000,   3},
```

For referring to an entry from this table CCD often uses the member elemRef of the table melem. An example reference is:

```
spare[melem[eRef].elemRef].bSize.
```

The last line of the table:

```
/*65535*/ { 0x00000000,   0},
```

can help to find the error source whenever a programmer uses an invalid index for referring to a spare IE.

5.7 melem.cdg

This table contains specific parameters needed for composing an IE. The format of the entries is given on the top of the list.

```
/* idx codingType optional extGroup repType calcIdxRef maxRep structOffs ident elemType ref */
```

Coding types:
The parameter **codingType** is necessary to choose the appropriate CCD encoding/decoding functions. Valid types (e.g. ASN1) are listed in the files **mconst.cdg** of each project or in **ccd_codingtypes.h** of the CCD software distribution.

Optional IE:
If the flag **optional** is set to **1**, the presence of the IEs depends on some parameters or conditions. This makes CCD check the conditions while encoding/decoding such an IE. CCD uses the expressions printed in the table calc by CCDGEN. The member **optional** is 1 in the following cases:

1. The IE belongs to an extended octet group. In this case the parameter **extGroup** can be one of the characters: '+', '-', '*', '!', '#' and ' '. The characters '+', '-' and '*' signify respectively the first, last and the single octet of an extended group. For the middle IEs shorter than 7 bits **extGroup** is set to ' '. Note that this character is also used for IEs that do not belong to an extended octet group. The example IEs are given in the following lines.

```
/* idx          extGroup          maxRep           elemType      */
/*      codingType     repType          structOffs              ref */
/*          optional       calcIdxRef          ident                */
/*  146*/ {   0, 1, '+', ' ',    0,    0,    0, 0xFFFF, 'S',    15 },
/*  147*/ {   0, 1, ' ', ' ',    0,    0,    0, 0xFFFF, 'V',    82 },
/*  148*/ {   0, 1, '*', ' ',    0,    0,    2, 0xFFFF, 'V',    93 },
```

2. The IE is specified by its tag value (IEI). In this case, the parameter **ident** is set to a value other than 0xFFFF.

```
/*  281*/ {   2, 1, ' ', ' ',    0,    0,    1, 0xD   , 'V',   111 },
/*  282*/ {   7, 1, ' ', ' ',    0,    0,    3, 0x4   , 'C',    21 },
```

3. The IE is specified to be an optional ASN.1 element to be encoded with Packed Encoding Rules. This can be for example a SEQUENCE, a CHOICE or an INTEGER element.

4. The presence of the IE depends on the fulfilling of some conditions. In this case the parameter **calcIdxRef** is not 0xFFFF any more. As a result the parameters **numCondCalcs** and **condCalcRef** in the calcidx table are set to other values than 0 and 65535, respectively. The parameter numCondCalcs is the number of calculation steps for a UPN calculator. The index condCalcRef refers to the entry for the first calculation step in the table calc. See also “online calculations” below.

Online calculations:
In the runtime CCD may need to check presence conditions, carry out prologue expressions or calculate length of an array. The required instructions are printed by CCDGEN in the file **calc.cdg**. The reference values and flags are printed to **calcidx.cdg**. For each IE with a **calcIdxRef** equal to 0xFF no instructions are given in the calc table. Otherwise **cacIdxRef** refers to the appropriate entry in the **calcidx** and from there to the appropriate instructions in the calc table.

In calcIdx information appears in pairs (reference and number of step): conCalRef together with numConCal, prolStepRef with numProlStep and repCalRef with numRepCal.

There is a complicated type of optional IEs to which "mob_ident" belongs. They need a few additional calculations compared with the usual optional IEs. For mob_ident not only numCondCalcs is different from 0 but also **numPrologSteps** (in the calcidx table).

Long name	short name	ref	bit len	Ctrl
Type of identity	ident_type		3	(GETPOS,:,4,+,:,1,+,SETPOS)
Odd/ Even indicac-tion	odd_even		1	(SETPOS)
Identity digit	ident_dig		4	(SETPOS) {ident_type # ID_TYPE_NO_IDENT AND ident_type # ID_TYPE_TMSI} BCDODD [0..16]
Spare	.1111		4	(:,SETPOS,8,+){ident_type = ID_TYPE_TMSI}
TMSI	tmsi		32	(SETPOS) { ident_type = ID_TYPE_TMSI} [.32]
Dummy	dmy		4	(SETPOS) {ident_type = ID_TYPE_NO_IDENT} [0..16]

```
/*  453*/ {   0, 0, ' ', ' ',   25,    0,    0, 0xFFFF, 'V',   163 },
/*  454*/ {   0, 0, ' ', ' ',   26,    0,    1, 0xFFFF, 'V',   160 },
/*  455*/ {  13, 1, ' ', 'i',   27,   16,    2, 0xFFFF, 'V',   153 }, -> ident_dig
/*  456*/ {   0, 1, ' ', ' ',   28,    0,    0, 0xFFFF, 'S',    38 }, -> 1111
/*  457*/ {   0, 1, ' ', 'b',   29,   32,   23, 0xFFFF, 'V',   164 }, -> tmsi
/*  458*/ {   0, 1, ' ', 'i',   30,   16,   36, 0xFFFF, 'V',   165 }, -> dmy
```

The numbers 25 to 30 refer to entries for conditions, prologues and size calculations in calcidx:

```
                ConCalRef,  numConCal,  prolStepRef,  numProlStep,  repCalRef,  numRepCal
/*   25*/ {  0,  65535, 8,   25,  0,     65535 }, <- prologue
/*   26*/ {  0,  65535, 1,   33,  0,     65535 }, <- prologue
/*   27*/ {  7,     34, 1,   41,  0,         0 }, <- condition, prologue, repetition
/*   28*/ {  3,     42, 4,   45,  0,     65535 }, <- condition and prologue
/*   29*/ {  3,     49, 1,   52,  0,        32 }, <- condition, prologue and repetition
/*   30*/ {  3,     53, 1,   56,  0,         0 }, <- condition, prologue and repetition
```

Since the last four elements are conditional, the field "optional" is set to 1 for them.

Bit, byte and element arrays:
There are IEs built of a series of values for a repeated variable. For theses IEs the parameter **repType** is no more set to ' '. The possible characters for a GSM standard IE are then '**i**', '**c**', '**v**' and '**b**,' which abbreviate respectively interval, constant, variable and bit field.

For elements encoded by PER '**C**', '**j**' and '**J**' have been added to this list. A repType of '**C**' specifies an array of an IE with the ASN.1 type BITSTRING. This is a bit string of constant size. Bit strings of variable size are given by **repType = 'J'**. Variable sized arrays of other ASN.1 PER types have **repType = 'j'**. Fixed sized arrays of other ASN.1 PER types have **repType = 'c'**.

If the number of repeats needs to be calculated the parameter **calcIdxRef** is different from **0**. If the message description gives a maximum number for the repeats the structure member **maxRepeat** is different from **0**. The four mentioned categories of repeated variables are discussed below. The corresponding parameters are underlined in each example.

For **repType = 'i'** the number of repeats belongs to a known interval. The upper limit of the interval is given by **maxRepeat**. It is **32** for the example IE "num" from CC.doc.

```
/*  206*/ {  14, 0, ' ', 'i',    7,   32,    4, 0xFFFF, 'V',    99 },
```

long name	short name	Ref	bit len	Ctrl
Number digit	num		4	BCDEVEN[0..32]

For **repType = 'c'** the number of repeats are known and constant. So **maxRepeat** is set to this value and **numRepCalcs** is **0**. The example IE "mcc" from MM.doc is made of three BCD numbers so the **maxRepeat** is set to **3** for it.

```
/*  447*/ {  15, 0, ' ', 'c',   23,    3,    0, 0xFFFF, 'V',   158 },
```

Long name	short name	ref	bit len	ctrl
Mobile Country Code	mcc		4	BCDEVEN[3]

For **repType = 'v'** the number of the repeats depends on the value of a variable. Thus in the **calcidx** table **numRepCalcs** is no more set to **0**. And the index **repCalcRef** refers to the entry for the first calculation step in the table calc. The example IE is "allo_bmp7" from RR.doc. It is a variable field the lenght of which depends on the variable allo_len7.

```
/*  788*/ {   0, 0, ' ', 'v',  110,  127,    2, 0xFFFF, 'V',   189 },
```

long name	short name	ref	bit len	type	ctrl
Blocks Or Block Periods	blp	6.21	1		
Allocation Bitmap Length	allo_len7	6.10	7		
Allocation Bitmap	allo_bmp7	6.9	1		[allo_len7..127]

The function ccd_calculatorep() of CCD will need only a read operation on the variable allo_len7. Therefore **cacIdxRef** is not 0xFF and **numRepCalcs = 1**. Using **repCalcRef** CCD looks at the appropriate entry of the table calc and finds there a read operation on the element of index 787 (= 0x313). And this element is nothing but the variable allo_len7.

For **repType = 'b'** the IE is a bit array of variable length. The maximum length (maxRepeat) is either given by a constant or by a number. An example for this case is the IE "non standard facilities" which is a bit field from **1** bit up to **N** bits, where **N** should be read from the constant value **MAX_NSF_LEN**.

```
/* 1589*/ {   0, 0, ' ', 'b',  254,  720,    4, 0xFFFF, 'V',   480 },
```

,	short name	ref	Ref	Pres	len	ctrl
Facsimile control field	fcf			M	1	
Non standard facilities	non_std_fac			M	1-N	[.MAX_NSF_LEN]

An instructive example is the bit field ussdString, which is the value part of an ASN.1 element encoded with Basic Encoding Rules:

ID	long name	short name	type	ref	ctrl	Len
0x04	Unstructured SS data coding scheme	ussdDataCodingScheme	ASN1	6.42		3
0x04	Unstructured SS data string	ussdString	ASN1	6.43	[.0..MAX_USSD_STRING]	2-162

Here the upper limit of the bit field should be given as the number of bits and not octets. And the value given for the constant MAX_USSD_STRING should be the same as the bitlen given in the section 6.43 for the basic element ussdString, currenlty 1280.

Location of an IE in the C-structure:
If the IE belongs to a structured IE, the variable **structOffs** will be different from **0** and refers to its place within the whole composition.

Tag or IE identifier:
If the IE has an information element identifier, the parameter **ident** will be different from **0** and containing the IE identifying number.

Spare, basic or structured element:
The member **elemType** can be one of the characters **V**, **S, C** and **U**, which abbreviate respectively variable, spare, composition and union. Depending on the element type the parameter **ref** must be interpreted as an index referring to the entry in **mvar**, **spare** or **mcomp** table.

5.8 mcomp.cdg

This table contains a few elementary parameters to specify an IE. The format of the entries is given on the top of the list.

```
/* idx name lnameRef cSize bSize numElems elemRef */
/*  0*/ { "aux_states"            ,  374,    4,    7,    3,     0 },
/*  1*/ { "bearer_cap"            ,  375,   74,   98,   37,     3 },
/*  2*/ { "bearer_cap_2"          ,  376,   74,   98,   37,    40 },
```

The variable **name** is the short name as given in the corresponding message description catalogue. The variable **lnameRef** longNameRef is an index referring to the entry for this IE in the table **mstr**. For messages **lnameRef** is set t0 0 because there is no entry for messages in mstr.

The number of bits used for an IE is stored to **bSize**. The space (in bytes) needed in form of a C-structure is given by **cSize**. For optional variables an additional byte is dedicated to valid flag. For example the IE "aux_states" from CC.doc has two optional variables: **hold** and **mpty**. The corresponding C-structure in m_cc.h looks like this:

```
typedef struct
{
  UBYTE           v_hold;         /*<  0> valid-flag                   */
  UBYTE           hold;           /*<  1> Hold auxiliary state         */
  UBYTE           v_mpty;         /*<  2> valid-flag                   */
  UBYTE           mpty;           /*<  3> Multi party auxiliary state  */
} T_aux_states;
```

Thus the **cSize** is 4 for this IE.

The parameter **numElems** gives the maximum number of sub IEs that the structured IE can contain. The variable **elemRef** is an index referring to the entry in the table **melem**.

5.9 mmtx.cdg

This table contains valid (and invalid) reference numbers for all possible (and impossible) IEs of a GSM application. The reference numbers are indexes referring to an entry in the table **mcomp**. In order to signify the invalid reference numbers for invalid IEs the entry here will be 65535. The format of the entries is given on the top of the list.

```
/* entity  msg_type  up   down */
/*[0000]*/
          /*[0000]*/ 65535,65535,
          /*[0001]*/    28,   27,
          /*[0002]*/ 65535,   30,
```

At the first look the table seems to be a simple byte field. In fact the table is built to be 3 dimensional. The 3 dimensions are:

1) the number of entities using CCD,
2) the number of message IDs and
3) the number 2 (uplink, downlink).

Therefore CCD will refer to an element of the table in this way:
mmtx[entity][message_type][direction].

Once we have the index from this table we can find more about the message through the **mcomp** table. From there we can find the details for composing its sub IEs via the table **melem**. That is why for coding a structured IE the function is called with only one parameter:

```
ccd_encodeComposition (mmtx[entity][theMsgId][direction]);
```

5.10 calc.cdg

This table contains parameters to specify the calculation steps for the UPN calculator. Each entry of the table represents one step. The format of the entries are given on the top of the list:

```
/* idx operation operand */
/*  0*/ { 'G', 0x00000000 },
/*  1*/ { ':', 0x00000000 },
/*  2*/ { 'P', 0x00000004 },
```

The meaning of the parameter **operand** depends on the character shown by the parameter **operation**. Possible characters for the member operation are:

operation	Meaning of operation	Role of **operand**
P	Push a constant on the stack	Constant number to read
R	Push the content of a C-structure variable on the stack	Index for the table melem
S	Get the upper element from the stack an set the position of the bit stream pointer to this value	Nothing
G	Push the position of the bit stream pointer on the stack	Nothing
:	Duplicate the upper element on the stack	Nothing
^	Swap the upper two elements of the stack	Nothing
+ - * /	Arithmetic operations	Nothing
& |	Bit operations: AND and OR	Nothing
A O X	Logical operations: AND, OR and XOR	Nothing
= # < >	Numerical comparisons	Nothing
K	Keep a value in the KEEP register	Index for keep register
L	Intended to copy the L part of a TLV element from the KEEP register[0]	Nothing
T	Take a value from the KEEP register and push it on the UPN stack	Index for keep register
C	Compare the value on the UPN stack with the one stored in the KEEP register and push the higher value in the KEEP register.	Index for keep register
Z	Used to mark presence of an address information part error label	Nothing
D	Used to mark presence of a distribution part error label	Nothing
N	Used to mark presence of a non distribution part error label	
M	Used to mark presence of a message escape error label	
I	Used to mark presence of an ignore error label	
l	Take a value from the CCD STO register and push it on the UPN stack	Index for STO register
s	Store a value in the CCD STO register	Index for STO register

CCD uses the UPN calculator when it checks a condition or when it reads the value of a variable or constant.

5.11 Example

In the previous sections we learned two kinds of formatted information:

1) Tables in the air message description files which are WINWORD documents
2) CCD tables currently in the output files of CCDGEN which are plain text files *.cdg

Now let us follow the information pieces of an example message from RR.doc into the CCD tables. The example message is the first message in that file: “additional assignment”.

Definition:

long name	short name	ID	direction
Additional assignment	d_add_assign	0b00111011	downlink

Elements:

ID	long name	short name	ref	ref [1]	pres	type	len
	Message Type	msg_type	6.60	10.4.	M	V3	1
	Channel Description	chan_desc	5.6	10.5.2.5	M	V	3
0x72	Mobile Allocation	mob_alloc	5.17	10.5.2.21	C	TLV	3-10
0x7C	Starting Time	start_time	5.27	10.5.2.38	O	TV	3

The entry for “additional assignment” in the table **mcomp** is

```
/*  266*/ { "D_ADD_ASSIGN",     0,   40,   148,    4,   969 },
```

This refers to the 969th entry in **melem** table. That entry is in turn:

```
/*  969*/ {   4, 0, ' ', ' ',    0,    0,    0, 0xFFFF, 'V',   302 },
```

This refres to the 302th entry in **mvar** table:

```
/*  302*/ { "msg_type", 1094,    8,   1, 'B',   0, 65535 },
```

This does not refer to an entry in the **mval** table since valDefRef = 0, 65535. The reason is that the values for msg_type (message Ids) are given in **mconst**. All other variable values are given in **mval**. Obviously while coding/decoding 0x3b (= 0b00111011) must be written/read for message type. Note that CCD needs no more information about this variable than its bit size. It also needs not to look for this value in any table. It reads the value from the buffer given by the entity.

At this point we have followed the trace of the first line (msg_type) in the table above.

Now let us look at the second line about chan_desc. This information element is not so simple as msg_type. The tabular description of this IE in the chapter „structured elements“ of the message description catalogues looks like this:

Elements:

long name	short name	ref	bit len	ctrl
Channel type and TDMA offset	chan_type	6.32	5	
Time Slot	tn	6.101	3	
Training Sequence Code	tsc	6.104	3	
Hopping	hop	6.51	1	
spare	.00		2	{hop=0}
Absolute RF Channel Number	arfcn	6.11	10	{hop=0}
Mobile Allocation Index Offset	maio	6.62	6	{hop=1}
Hopping Sequence Number	hsn	6.52	6	{hop=1}

The following describing entry:

```
/*  139*/ { "chan_desc_2",  1397,   12,    36,    8,   561 },
```

in **mcomp** refers to an entry in **melem** for the first variable of this group, namely chan_type.

```
/*  561*/ {   0, 0, ' ', ' ',    0,    0,    0, 0xFFFF, 'V',  220 },
```

The other variables follow this entry:

```
/*  562*/ {   0, 0, ' ', ' ',    0,    0,    1, 0xFFFF, 'V',   399 }, -> tn
/*  563*/ {   0, 0, ' ', ' ',    0,    0,    2, 0xFFFF, 'V',   412 }, -> tsc
/*  564*/ {   0, 0, ' ', ' ',    0,    0,    3, 0xFFFF, 'V',   275 }, -> hop
/*  565*/ {   0, 1, ' ', ' ',   42,    0,    0, 0xFFFF, 'S',    46 }, -> .00
/*  566*/ {   0, 1, ' ', ' ',   43,    0,    5, 0xFFFF, 'V',   179 }, -> arfcn
/*  567*/ {   0, 1, ' ', ' ',   44,    0,    8, 0xFFFF, 'V',   304 }, -> maio
/*  568*/ {   0, 1, ' ', ' ',   45,    0,   10, 0xFFFF, 'V',   276 }, -> hsn
```

We see that for all elements **codingType = 0** because they are simple values. For the last four variables **Optional = 1**. They are conditional variables. The UPN calculator will check the condition.
Following the four values 42-45 for **calcIdxRef** we find entries in **calcIdx,** which lead to entries in calc for online calculations.

```
/*   42*/ {  3,    57, 0, 65535,    0, 65535 },
/*   43*/ {  3,    60, 0, 65535,    0, 65535 },
/*   44*/ {  3,    63, 0, 65535,    0, 65535 },
/*   45*/ {  3,    66, 0, 65535,    0, 65535 },
```

In the run time CCD will read three lines (**numCondCalcs = 3**) beginning with the 57th line (**condCalcRef = 57**) from the table **calc** in order to check the condition for the element **arfcn**. And so on.

Now back to the first element. More about chan_type is to find in **mvar**, e.g. its bit size.

```
/*  220*/ { "chan_type",  873,    5,   1, 'B',  28,  571 },
```

Thus chan_type is 5 bits long and needs one byte in the C structures. This entry also refers to the (at least) 28 different valid values for chan_type listed in **mval**. The values are between 1 and 28 plus the default of „undefined“. They are to find in the 571th line of **mval**.

```
/*  571*/ {     0, 0, 0x00000001, 0x00000001}, <- 1
/*  572*/ {     0, 0, 0x00000002, 0x00000002}, <- 2
...
/*  595*/ {     0, 0, 0x00000019, 0x00000019}, <- 25
/*  596*/ {     0, 0, 0x0000001A, 0x0000001A}, <- 26 (27, 28 and 29 are no valid values)
/*  597*/ {     0, 0, 0x0000001E, 0x0000001E}, <- 30
/*  598*/ {     0, 1, 0x00000000, 0x00000000}, <- default
```

We see that these entries correspond to the value tables of the chapter basic elements in the air message catalogue RR.doc.

Values:

value	c-macro	comment
1	TCH_F	TCH/F + ACCHs
2	TCH_H_S0	TCH/H + ACCHs, subchannel 0
3	TCH_H_S1	TCH/H + ACCHs, subchannel 1

...

24	TCH_F_ADD_UNI1	TCH/F+ACCHs, additional unidirectional TCH/FD/SACCH/MD on timeslot n-1
25	TCH_F_ADD_UNI2	TCH/F+ACCHs, additional unidirectional TCH/FD/SACCH/MD on timeslot n+1, n-1
26	TCH_F_ADD_UNI3	TCH/F+ACCHs, additional unidirectional TCH/FD/SACCH/MD on timeslot n+1, n-1, n-2
30	TCH_F_ADD_BI_UNI	TCH/F+ACCHs, additional bidirectional TCH/F/SACCH/M and unidirectional TCH/FD/SACCH/MD on timeslot n+1, n-1
DEF		channel not defined

On the PC version of CCDDATA entries in mval refer also to the long names of the values, all collected in **mstr.cdg**. This name is important for example for any test application with good comments. The name of the first vlaue is given by:

```
/* 874*/ "TCH/F + ACCHs",
```

Similar considerations can be made for the IEs **mob_alloc** and **start_time**. Note that there is no additional entry in the **mcomp** or **melem** tables for **T** and **L** elements of a **TLV** type IE. The **T** part is embedded in the entry in **melem**:

```
/*  971*/ {   7, 1, ' ', ' ',    0,    0,   19, 0x72   , 'C',   173 },
```

and **L** part is a dynamic value which is calculated and written in the message according to each message structure.

6 Generated C-Code Header Files

6.1 CSN1 Coding

6.1.1 CSN1_S1

The CSN1_S1 coding type is an appropriate solution if an optional information element is described by the following example:
<something> ::= 0 | 1 <element>;

CCDGEN generates the following C-structure:

```
typedef struct
{
  U8        v_element;  /* Valid-Flag */
  T_element element;    /* Structured Element */
} T_something;
```

If <element> is a structured element you need another structured element definition table defining the internal element composition. With the assumption that <a>, <b> and <c> are structured elements too, CCDGEN generates the following C-structures:

<element> ::= <a> <b> <c>;

```
typedef struct
{
  T_a       a;          /* Structured Element a */
  T_b       b;          /* Structured Element b */
  T_c       c;          /* Structured Element c */
} T_something;
```

If <element> is a basic element you need another basic element definitions table. In this case the generated C-structure will be very simple; see the following example:

<element : bit(5)>

```
typedef struct
{
  U8        v_element;  /* Valid-Flag */
  U8        element;    /* Basic Element */
} T_something;
```

Required actions for encoding:

If the structured element <element> is supposed to be present in an emitted message stream the associated valid flag has to be set to “1” and the user has to assign an appropriate value to the C-structure of the struct element <element>. Calling ccd_codeMsg(..) / ccd_codeMsgPtr(..) performs the encoding process according to the generated ccddata tables.

Required actions for decoding:

Calling ccd_decodeMsg(..) / ccd_decodeMsgPtr(..) performs the decoding process of a received message stream according to the generated ccddata tables. The user has to analyse the valid flag to determine if element <element> is present. The further processing should depend on the information given by the valid flag.

6.1.2 CSN1_S0

The CSN1_S0 coding type is an appropriate solution if an optional information element is described by the following example:

<something> ::= 1 | 0 <element>;

CCDGEN generates the same C-structures like shown above (please, see C-structures belonging to coding type CSN1_S1).

6.1.3 CSN1_SHL

The coding type CSN1_SHL supports elements comprising of a single bit valid flag and a value part. Only if the valid bit is equal to H the value part follows. Otherwise (valid bit is equal to L) the value part is absent.

Example:

<something> ::= L | H <element>;

CCDGEN generates the same C-structures like shown above (please, see C-structures belonging to coding type CSN1_S1).

6.1.4 HL_FLAG

A HL_FLAG element consists of a single bit only. The decoded value will be 0 if the encoded value is L respectively 1 if the encoded value is H. This element enables support of a choice according to the following example:

<z> ::= { { L <a> } | { H <b> };

With the assumption that <a> and <b> are structured elements too, you will get by CCDGEN the following type definition:

```
typedef struct
{
  U8        flag;       /* Basic Element */
  U8        v_a;        /* Valid-Flag */
  T_a       a;          /* Structured Element a */
  U8        v_b;        /* Valid-Flag */
  T_b       b;          /* Structured Element b */
} T_something;
```

Required actions for encoding:

If the structured element <a> is supposed to be present in an emitted message stream the struct element <flag> has to be set to “0”. The user must set <v_a> to “1” and <v_b> to “0”; besides the user has to assign an appropriate value to the C-structure of the struct element <a>. Calling ccd_codeMsg(..) / ccd_codeMsgPtr(..) performs the encoding process according to the generated ccddata tables.

Required actions for decoding:

Calling ccd_decodeMsg(..) / ccd_decodeMsgPtr(..) performs the decoding process of a received message stream according to the generated ccddata tables. The user has to analyse at least one of the valid flags to recognize which element (<a> or <b>) is present. In this case alternatively the element <flag> can be used to distinguish between both cases. The further processing of the entries in the C-structure should depend on the information either given by the element <flag> or one of the valid flags.

6.1.5 CSN1_CONCAT

Truncated concatenation is a special sequence of components, which allows any of the concatenations starting with null and up to any number of components arranged in a definite sequence.

Example:

<z> ::= { < a > < b > < c > } //;

The sequence of components (truncated concatenation) must be handled as a structured information element associated with the coding type CSN1_CONCAT, which comprises all subcomponents.

With the assumption that <a>, <b> and <c> are structured elements too, you will get by CCDGEN the following type definitions:

```
typedef struct
{
  T_a       a;          /* Structured Element a */
  T_b       b;          /* Structured Element b */
  T_c       c;          /* Structured Element c */
} T_x;

typedef struct
{
  T_x       x;          /* Structured Element of coding type CSN1_CONCAT*/
} T_z;
```

As shown above a truncated concatenation may comprise components associated with the coding types which do not characterise optional elements inherently. In this case CCDGEN provides appropriate elements without associated valid flags in the C structures; although it should be possible to truncate the sequence. The absence of an element associated with this kind of coding type truncates the sequence.

So far some coding types (like tagged types, e.g. GSM3_TV) characterise optional elements inherently. If you describe a component in the message description by one of these coding types CCDGEN provides an appropriate element associated with a valid flag in the C structures while generating C header files. The value of the valid flag indicates the presence or absence of such an element.

Components concatenated with a CSN1 coding type cause these valid flags in the C header structure too. If you find a bit in the received message stream indicating optional values not included in the message (e. g. a CSN1_S1 element is represented by '0'), CCD will set the valid flag to zero. If this component belongs to a truncated concatenation the absence of the value does not truncate the sequence. See the example below:

```
<z>    ::=  { < a >     - TLV coded element
              0 | 1 < b >
              < c > } //;
```

With the assumption that <a>, <b> and <c> are structured elements too, you will get by CCDGEN the following type definitions:

```
typedef struct
{
  U8        v_a;        /* Valid-Flag */
  T_a       a;          /* Structured Element a */
  U8        v_a;        /* Valid-Flag */
  T_b       b;          /* Structured Element b */
  T_c       c;          /* Structured Element c */
} T_x;
```

```
typedef struct
{
  T_x       x;          /* Structured Element of coding type CSN1_CONCAT*/
} T_z;
```

In case of truncated concatenations neither the absence of a valid flag nor a valid flag set to zero is a definite indication of an element's absence. Therefore you need an aid to recognize how many components could be decoded out of a received message stream.

It is recommended to write a leading element of coding type NO_CODE in the message description which is used to count the existing elements of the truncated concatenation. In case of decoding CCD writes the number of decoded elements belonging to the truncated concatenation to the NO_CODE element. Example of the associated C-structure:

```
typedef struct
{
  U8        element_no; /* Basic Element of coding type NO_CODE */
  U8        v_a;        /* Valid-Flag */
  T_a       a;          /* Structured Element a */
  U8        v_a;        /* Valid-Flag */
  T_b       b;          /* Structured Element b */
  T_c       c;          /* Structured Element c */
} T_x;

typedef struct
{
  T_x       x;          /* Structured Element of coding type CSN1_CONCAT*/
} T_z;
```

In case of encoding CCD always encodes all elements belonging to the truncated concatenation.

If the truncated concatenation not finishing the message description is followed by any other elements you will have another message element characterising the bit length of the truncated concatenation. In this case the structured element item must be associated with a type modifier indicating a bit string.

Example:

< z > ::= <length : bit (6)>
< bit (val(length) + 1)
& { { < a > < b > < c > } // ! { bit ** = <no string> }

Generated C-Structure:

```
typedef struct
{
  U8        element_no; /* Basic Element of coding type NO_CODE */
  U8        v_a;        /* Valid-Flag */
  T_a       a;          /* Structured Element a */
  U8        v_a;        /* Valid-Flag */
  T_b       b;          /* Structured Element b */
  T_c       c;          /* Structured Element c */
} T_x;

typedef struct
{
  U8        length;     /* Length Information (Basic Element) */
  T_x       x;          /* Structured Element of coding type CSN1_CONCAT*/
} T_z;
```

Required actions for encoding:

In case of encoding CCD steps though the C-structure according to the instructions given by the ccddata tables and always encodes all elements belonging to the truncated concatenation.

If optional structured elements are supposed to be present in an emitted message stream their associated valid flag has to be set to “1” and the user has to assign an appropriate value to the C-structure of the particular element. All other struct elements of the C-structure must be filled with correct values. Calling ccd_codeMsg(..) / ccd_codeMsgPtr(..) performs the encoding process according to the generated ccddata tables.

Note:
If the truncated concatenation is associated with a length information the user is expected to determine the value of this length information without CCD support. So far CCD does not evaluate this length information according to length of encoded elements like it is done for TLV compositions. This feature will be a future enhancement.

Required actions for decoding:

Calling ccd_decodeMsg(..) / ccd_decodeMsgPtr(..) performs the decoding process of a received message stream according to the generated ccddata tables. If the message description provides a leading NO_CODE element the user can analyse this element to detect easily the number of received elements belonging to the truncated concatenation. Otherwise the user has to take appropriate actions to recognize which elements have been received. In case of existing valid flags it is very easy.

6.1.6 BREAK_COND

Decoding/encoding of the element Repeat_struct:
This element consists of a V component with a variable bit length and must be connected with a special condition. This condition has to be a simple value, which matches to the value range of BREAK_COND element itself.

This function performs a standard decoding for a given elem table entry. This means for non structured elements that 1 - n bits are read from the bitstream and write to a C-Variable in a machine dependent format. After decoding of the requested number of bits the resulting value will be compared with the constant given by the condition. In case of equality the global variable globs->continue_array is set to FALSE. This breaks decoding of the current superior composition and finishes the array.

```
typedef struct
{
   T_a          a;          /* Structured Element */
   U8           number;     /* Basic Element */
   U8           v_b;        /* valid-flag */
   U8           c_b;        /* counter */
   T_b          b[MAX_NUM]; /* Structured Element */
} T_rep_struct;

typedef struct
{
   U8           v_rep_struct;              /* valid-flag */
   U8           c_rep_struct;              /* counter */
   T_rep_struct ncp2_rep_struct[MAX_REPS]; /* Structured Element */
   U8           x;                         /* Basic Element */
} T_z;
```

6.1.7 CSN1_CHOICE1 and CSN1_CHOICE2

The coding types CSN1_CHOICE1 and CSN1_CHOICE2 support elements comprising of a flag and alternative value parts depending on this flag value. In case of CSN1_CHOICE1 only one bit is used as flag adressing two alternative value parts. If the flag bit is equal to “0” the first value part follows and the second is absent. Otherwise (flag bit is equal to “1”) the first value part is missing instead the second value part is present. In case of CSN1_CHOICE2 the flag is two bit long and it allows adressing of four alternative avlue parts.

Example (CSN1_CHOICE1):

```
<msg_ex>  ::= {  <x>
                 {{ 0 <a>} | { 1 <b>}}
                 <z>}
```

The coding type CSN1_CHOICE1 enables the choice construction

```
<y>       ::= { 0 <a>} | {1 <b>};
```

With the assumption that <x>, <z>, <a> and <b> are structured elements too, you will get by CCDGEN the following type definitions (alignment has met with no response):

```
typedef struct
{
  T_x       x;          /* Structured Element x */
  T_ctrl_y  ctrl_y;     /* (enum=32bit) controller for union /
  T_y       y;          /* Element will be a union type with controller */
  T_z       z;          /* Structured Element z */
} T_msg_ex;

typedef union
{
  T_a       a;          /* Structured Element a */
  T_b       b;          /* Structured Element b */
} T_y;

typedef enum
{
  choiceA_1 = 0x0,
  choiceA_2 = 0x1
}T_ctrl_y;
```

In case of coding type CSN1_CHOICE1 CCD reads one bit (in case of CSN1_CHOICE2 two bits) and writes the result to structure item ctrl_y. CCD determes "elemRef" depending on the value of ctrl_y and processes the union element according to table entry of "elemRef".

6.1.8 CSN1_S1_OPT

The CSN1_S1_OPT coding type is an appropriate solution if an optional information element is described by the following example:
<something> ::= null | 0 | 1 <element>;

The element value can be present, then it is preceded by the flag value 0. If the element is represented by the flag 1 only the value is absent! Besides the absence of the whole element is allowed.

This coding type is very similar to the coding type CSN1_S1, which doesn’t support the possibility of element lack. CSN1_S1_OPT is the appropriate solution if the specification demands a construction with a composition of coding type CSN1_CONCAT and an encapsulated CSN1_S1 element. This type increases the recursions level and therefor the runtime performance would be decreased.

CCDGEN generates the following C-structure:

```
typedef struct
{
  U8        v_element;  /* Valid-Flag */
  T_element element;    /* Structured Element */
} T_something;
```

If <element> is a structured element you need another structured element definition table defining the internal element composition. With the assumption that <a>, <b> and <c> are structured elements too, CCDGEN generates the following C-structures:

<element> ::= <a> <b> <c>;

```
typedef struct
{
  T_a       a;          /* Structured Element a */
  T_b       b;          /* Structured Element b */
  T_c       c;          /* Structured Element c */
} T_something;
```

If <element> is a basic element you need another basic element definitions table. In this case the generated C-structure will be very simple; see the following example:

<element : bit(5)>

```
typedef struct
{
  U8        v_element;  /* Valid-Flag */
  U8        element;    /* Basic Element */
} T_something;
```

Required actions for encoding:

If the structured element <element> is supposed to be present in an emitted message stream the associated valid flag has to be set to “1” and the user has to assign an appropriate value to the C-structure of the struct element <element>. Otherwise the associated valid flag has to be set to “0”. Calling ccd_codeMsg(..) / ccd_codeMsgPtr(..) performs the encoding process according to the generated ccddata tables.

Required actions for decoding:

Calling ccd_decodeMsg(..) / ccd_decodeMsgPtr(..) performs the decoding process of a received message stream according to the generated ccddata tables. The user has to analyse the valid flag to determine if element <element> is present. The valid flag may be left out (either the encoded stream fills the specified message buffer or a superordinate length information indicates the structure’s end). The further processing should depend on the information given by the valid flag.

6.1.9 CSN1_S0_OPT

The CSN1_S0_OPT coding type is an appropriate solution if an optional information element is described by the following example:

<something> ::= null | 1 | 0 <element>;

CCDGEN generates the same C-structures like shown above (please, see C-structures belonging to coding type CSN1_S1_OPT respectively CSN1_S1_OPT).

6.1.10 CSN1_SHL_OPT

The coding type CSN1_SHL_OPT supports elements comprising of a single bit valid flag and a value part. Only if the valid bit is equal to H the value part follows. Otherwise (valid bit is equal to L) the value part is absent. Besides the absence of the whole element is allowed.

Example:

<something> ::= null | L | H <element>;

CCDGEN generates the same C-structures like shown above (please, see C-structures belonging to coding type CSN1_S1_OPT respectively CSN1_S1_OPT).

6.2 Special Coding Types

6.2.1 S_PADDING

Padding bits does not appear in the generated C-structure!

6.2.2 S_PADDING_0

Padding bits does not appear in the generated C-structure!

6.2.3 FDD_CI, TDD_CI

The decoder function for the type FDD_CI (TDD_CI) decodes two information elements. First it decodes five bits as NR_OF_FDD_CELLS (NR_OF_TDD_CELLS). Then it decodes the list of fdd_ci_parameters.

For NR_OF_FDD_CELLS=0 or if NR_OF_FDD_CELLS is equal to or greater than 17 (21 for TDD_CI), the function returns immediately. CCD then continues decoding the next IEs.

If the number is small enough, it calculates the bit size of the parameter list and decodes them.

If the calculated bit size exceeds the left space in the message buffer, CCD will report an error on ERR_ELEM_LEN. Because of the risk of buffer overwriting and since the reason could be a truncated message, CCD will break its decoding activities right after reporting ERR_ELEM_LEN.

NOTE: Decoded parameters are written in an array of U16 words. It is the task of the entity to extract the subelements from each parameter by using bit manipulation operators. Subelements of fdd_ci_parameters are Scrambling Code (bits 1-9) and Diversity bit (bit 10). Subelements of tdd_ci_parameters are Cell Parameter (bits 1-7), Sync Case (bit 8) and Diversity bit (bit 9).

```
typedef struct {
  U16 tdd_arfcn
  U8  tdd_indic_0;               /* information 0 indicator*/
  U8  c_tdd_cell_information;    /* counter */
  U16 tdd_cell_information[21];  /* cell parameter, */
} T_tdd_ci_cmp;

typedef struct {
  U16 fdd_arfcn
  U8  fdd_indic_0;               /* information 0 indicator*/
  U8  c_fdd_cell_information;    /* counter */
  U16 fdd_cell_information[17];  /* cell parameter, */
} T_fdd_ci_cmp;
```

6.3 Some tricky descriptions for particular message elements

6.3.1 Error Labels

Except of the error label MESSAGE ESCAPE all other error labels have not any impact on the generated C-structure. The error label Message Escape should be concatenated with an element of coding type NO_CODE. Therefore this construction causes the following two lines in the type definition:

```
  U8        v_no_code;  /* valid-flag */
  U8        no_code;    /* intended to use with coding type NO_CODE */
```

6.4 How to express non-standard length information

CCD handles elements of variable length which can not be described standard LV-coding types as a bit string independent of the internal structure of <b>. The element <b> may be a basic or a structured element. The length of this bit string must depend on another basic IE declared before.

CCDGen does not process a structured IE given as bit string as an array type. In the generated header file you will not find any counter for it. The comment in the C-declaration is preceded by “BIT STRING:” to underline the special character of this structure.

```
typedef struct
{
  U8        length;     /* Length Information (Basic Element) */
  T_b       b;          /* BIT STRING: Structured Element of type T_b */
} T_z;
```

Required actions for encoding:

The length of the structured element <b> must be calculated and assigned to the C-structure by the CCD user in an appropriate manner. So far it is not possible to determine this length value depending on its nested encoded elements. Calling ccd_codeMsg(..) / ccd_codeMsgPtr(..) performs the encoding process according to the generated ccddata tables. Before CCD implies any encode actions on an structured IE given as bit string, it calculates the bit position at the end of the bit string. After encoding of known sub elements CCD jumps to the calculated position and continues its work on the next information element.

Required actions for decoding:

Calling ccd_decodeMsg(..) / ccd_decodeMsgPtr(..) performs the decoding process of a received message stream according to the generated ccddata tables. Before CCD implies any decode actions on an structured IE given as bit string, it calculates the bit position at the end of the bit string. This means the bit position after the last bit of the bit string structure. After decoding of known sub elements CCD jumps to the calculated position and continues its work on the next information element.

If the calculated bit position exceeds the message length or the wrapping structured IE, CCD generates a warning with the error code ERR_MAX_REPEAT. The error code ERR_BITSTR_COMP is reported as warning if decoding or encoding actions results to a higher value for bit position than the expected value. If the calculated_bit_position exceeds its theoretical maximum (upper limit of the range) its value will be cut to work within the allowed limits. CCD reports a warning.

The limit on bit size is temporarily set to a value based on the bit string length. After decoding of the whole bit string CCD checks if any mandatory IE is left not decoded. If this is true CCD reports an error (error code: ERR_MAND_ELEM_MISS) followed by an abortion of decoding action.

7 How to Call GTC Tools

[TBD]

Appendices

A. Examples

B. Acronyms

DS-WCDMA Direct Sequence/Spread Wideband Code Division Multiple Access

C. Glossary

International Mobile Telecommunication 2000 (IMT-2000/ITU-2000) Formerly referred to as FPLMTS (Future Public Land-Mobile Telephone System), this is the ITU's specification/family of standards for 3G. This initiative provides a global infrastructure through both satellite and terrestrial systems, for fixed and mobile phone users. The family of standards is a framework comprising a mix/blend of systems providing global roaming. <URL: http://www.imt-2000.org/>

D. Syntactic metanotation

Table Appendix D 1 defines the metanotation used to specify the form of grammar similar to BNF used in this document:

::=	is defined to be
abc xyz	abc followed by xyz
|	alternative
[abc]	0 or 1 instances of abc
{abc}	0 or more instances of abc
{abc}+	1 or more instances of abc
(...)	textual grouping
abc	the non-terminal symbol abc
abc	a terminal symbol abc
"abc"	a terminal symbol abc

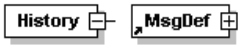
Table Appendix D 1: Syntactic Metanotation

E. Legend of Symbols Documenting the XML Format

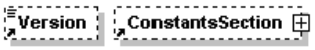
Some symbols are used within the XML format description for SAP and AIM, which may not be familiar for all readers. The following should give an overview about these symbols and what their meaning is. To clarify this, these are only symbols used throughout this document and do not represent the presentation in an editor for the new format.

Represents a simple element which could contain alphanumerical data. The name of the element (refers to the XML tag name) is shown in the centre of the box. This kind of element does not have child elements. They are more or less child elements itself. The simple box indicates that this element occurs only once.

Represents a simple element which does not contain any data. Nevertheless this element could have attributes which can carry information. Without attributes, the existence of the element is the information itself. The name of the element (refers to the XML tag name) is shown in the centre of the box. This kind of element does not have child elements. They are more or less child elements itself. The simple box indicates that this element occurs only once.

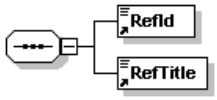


Represents a parent element which does not contain any data but has one or more child elements. The name of the element (refers to the XML tag name) is shown in the centre of the box. The plus/minus sign in the small box indicates whether the child elements are unfolded or hidden (similar to the graphical representation of a directory structure). The small arrow does not have a meaning. The simple box indicates that this element occurs only once.

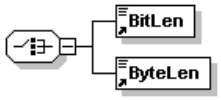


If a child or parent element is optional in its use then the border of the element box is drawn with a dashed line. This means that this element could occur once or never.

If a child or parent element could appear more than once a multiple box will be displayed. A small indication at the bottom of the box shows if the minimum occurrence of the element is zero (optional) or if at least one element has to be present.



If child elements of a specific parent element have to be in a certain order, then this is indicated by a diamond-shaped symbol showing a string of dots in its centre. The order of appearance for the child elements is from top to bottom (e.g. first *RefId* and second *RefTitle*). The plus/minus sign in the small box indicates whether the child elements are unfolded or hidden (similar to the graphical representation of a directory structure).



If only one child element out of a selection of several elements could occur for a parent element, then this is indicated by a diamond-shaped symbol showing a small switch in its centre (e.g. either *BitLen* or *ByteLen* could be the child element). The plus/minus sign in the small box indicates whether the child elements are unfolded or hidden (similar to the graphical representation of a directory structure).

F. Index

G. Table of Figures

H. Table of Tables