Technical Document

GSM PROTOCOL STACK

GPF

CCD_USERGUIDE.DOC

CCD USERS´ GUIDE

TI INTERNAL TECHNICAL DOCUMENT

Document Number:	06-03-20-SHL-0002
Version:	0.4
Status:	Draft
Approval Authority:	
Creation Date:	2002-May-31
Last changed:	2015-Mar-08 by SIJ
File Name:	CCD_userguide.doc

Important Notice

Change History

Date	Changed by	Approved by	Version	Status	Notes
2002-May-31	SIJ		0.1		1
2002-Oct-23	SIJ		0.2		2
2003-May-20	XINTEGRA		0.3	Draft	
2003-Aug-19	SIJ		0.4	Draft	3

Notes:

1. Initial version
2. Updated to CCD II
3. New document number; Updated info about melem and bit fields

Table of Contents

List of Figures and Tables

List of References

[ISO 9000:2000] International Organization for Standardization. Quality management systems - Fundamentals and vocabulary. December 2000

1.1 Abbreviations

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

1.2 Terms

Entity: Program which executes the functions of a layer

Message: A message is a data unit which is transferred between the entities of the same layer (peer-to-peer) of the mobile and infrastructure side. Message is used as a synonym to protocol data unit (PDU). A message may contain several information elements.

Primitive: A primitive is a data unit which is transferred between layers on one component (mobile station or infrastructure). The primitive has an operation code which identifies the primitive and its parameters.

Service Access Point: A Service Access Point is a data interface between two layers on one component (mobile station or infrastructure).

Information Element An information element is part of messages of the air-interface. There are mandatory, optional and conditional information elements. An IE consists of parameters.

2 Introduction

In the world of GSM there are messages transmitted over the air-interface. For GSM 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. Data in GSM protocol stack entities are normally hold in c-structures.

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 (e.g. Motorola MC680XX) 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.

Encoding and decoding for the TI GSM Protocol Stack is performed by the CCD. Additionally there are some functions included to code and decode simple data types like byte and long.

This document describes the functions which are used for coding and decoding. Also the data types, constants and parameters for the functional specification are introduced.

This document is a **CCD Programmer's Guide**. Yet it can also serve as a CCD users' guide, because the users of CCD need to know how it handles the message data. This guide is based on the assumption that the user is familiar with the basic operation principles of mobile phones.

2.1 Overview

The second chapter of this document is dedicated to the fundamental information about encoding/decoding of data for GSM applications. Most of the needed coding types are discussed. Coding rules are also dependent on the content of the message description files. It is described how the content of tables in those files can affect the work of CCD.

The third chapter gives a detailed description of the modules in CCD. While the first two sections can be useful for any user of CCD, the later sections are supposed to help only a smaller group of users. That means those of them who plans to understand, change or enhance CCD.

The fourth chapter is a technical note on CCD as a software distribution. It shows how to build CCD object or library files.

Please note that this document is permanently under construction. The author appreciates any feedback from its readers.

3 Fundamentals

3.1 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.

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

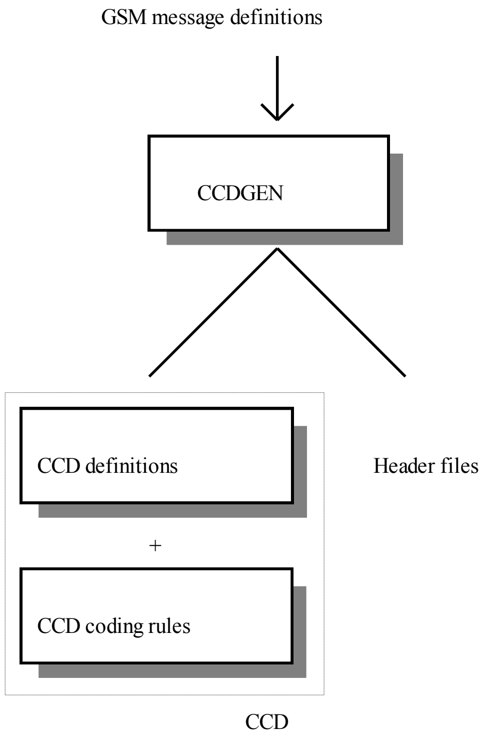
The interface to the applications, i.e. the Protocol-Stack components, is very simple, consisting of an encoding and a decoding function. Thus, all encoding or decoding is carried out by a single function call to CCD. The interface offers also functions to retrieve information on errors occurred while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

It is also possible to use CCD database to represent messages in readable form. Applications using CCD are test systems and tools for analysing signalling.

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. **At run-time CCD does not work with primitives**. 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 type 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. This is carried out by CCD 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.

The CCD, as a coder/decoder, has two parts located on the target system. The coding rules for different coding types and the data about the supported messages and their components.



For a better understanding of the CCD functionality the following sections handle fundamental aspects of message structures and their elements. A few coding types are described and depicted in details. Also the structure of the so-called message description documents are discussed in this chapter. The description is completed with an example in the last section.

3.2 Coding Types

CCD uses different functions for coding/decoding of the different types and formats of the information elements. Next subsection contains definitions for the type and format of a standard IE. Until then we use the word “type” in the context of “coding type”. At the beginning of each message description catalogue (usually msg/*.doc files) there is a table of used coding types. Type names in CCD and in message description catalogues are just a little different from each other. For example the type GSM1_V will change into CCDTYPE_GSM1_V in ccd source code. Until today the following types have been considered by CCD.

1) For standard information elements: GSM1_V, GSM1_TV, GSM2_T, GSM3_T, GSM3_TV, GSM4_TV, GSM4_TLV, GSM5_V, 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_SHL, S_PADDING and GSM7_LV.

In a mixed or nested IE several coding types can be used. A good example is BCDODD which is sometimes used for basic elements within a standard IE. The next subsections will give a short description of the mentioned coding types.

3.2.1 Standard Information Elements

Not all but most of the L3 messages have the standard structure described in the GSM documentation 04.07. 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**.

The imperative part	The non-imperative part
msg header | first IE . . . last IE	first IE . . . last IE

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 will be superfluous here.

A standard IE may have the following parts, in that order:
- an information element identifier (IEI);
- a length indicator (LI);
- a value part.

They are also known as Type (T), Lenght (L) and Value (V). In the comments of the CCD source code the author uses Tag instead of Type to refer to the IE Identifier. If an optional IE is not present in a message instance, none of the three parts is present. A standard IE has one of the formats shown in table:

Format	Meaning	IEI?	LI?	V?
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

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

If 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.

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 may be further structured into parts, called fields.

Totally four categories of standard IEs are defined:
1) IEs of format V or TV with value part consisting of 1/2 octet (type 1) which are known as GSM1_V and GSM1_TV in CCD tables.

Type 1 IE of Format V

8 4 1

- - - -	value part

Type 1 IE of Format V

8 4 1

value part	- - - -

Type 1 IE of Format TV

8 4 1

IEI	value part

-2) IE of format T with value part consisting of 0 octets (type 2) which is known as GSM2_T in the CCD tables.

Type 2 IE of Format T

8 4 1

IEI

3) IEs of format V or TV with value part that has fixed length of at least one octet (type 3) which are known as GSM3_V and GSM3_TV in the ccd tabls.

8 4 1

value part	Octet n
value part	Octet n+1
.	
.	
.	
value part	Octet n+k

Typ 3 IE des Formats V

8 4 1

IEI
value part
.
.
.
value part

Typ 3 IE des Formats TV

4) IEs of format TLV or LV with value part consisting of zero, one or more octets (type 4) which are known as GSM4_TV, GSM4_TLV, GSM1_ASN and GSM1_ASN_NULL in the CCD tables. In a IE of type 4 (LV, TLV, ASN or ASN_NULL) the value part consists of zero, one, or more octets. If present, its IEI has one octet length.

In case of LV the value part of the IE is mandatory. In case of TLV the length of the value part must be calculated by the real time application. 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.

	8 4 1	8 4 1	8 4 1
Octet n	LI	IEI	IEI
	value part	LI	LI = 0x0080
	value part	value part	value part
	.	.	.
	.	.	.
	.	.	.
	value part	value part	0 0 0 0 0 0 0 0
			0 0 0 0 0 0 0 0

Octet n+1
Octet n+2
.
.
.
Octet n+k

Type 4, Format LV	Type 4, Format TLV or ASN.1 BER	ASN.1 BER with EOF (End Of Content octets)

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 is an extension for the type 4 IEs of format TLV which is defined by CCD as a fifth type. It is called GSM5_TLV and can have two octets for the LI. For LIs above 127 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 consists of an octet.

Another extended type is called GSM5_V. It is used for writing of raw or undecoded bits which has 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 following table shows by which functions each coding type of standard IEs is actually processed.

coding type	decoding function
GSM1_V, GSM3_V	cdc_STD_decode
GSM5_V	bf_readBitChunk
GSM1_TV, GSM2_T, GSM3_TV, GSM4_TLV, GSM5_TLV GSM1_ASN, GSM1_ASN_NULL GSM4_LV	cdc_TLV_decode

3.2.1.1 Value Part of a standard IE

The value part of a standard IE can be another IE, a few variables or a few spare bits. Often spare bits are a serie of zeros which help to fill up an octet. All these three types can be mandatory or optional. One way to show a variable in the value is using a Binary Code Decimal (BCD) number.

3.2.1.2 BCD numbers

There are numbers in GSM world which should be handled digit for digit, e.g. the mobile identity. In these cases GSM uses Binary Code Decimal numbers. The smallest unit of a BCD number is called nibble which is made up of four bits. Each nibble represents a digit of a decimal number. For example the number 5,319 is shown as:

5	3	1	9
0101	0011	0001	1111

At the beginning of such arrays the first digit may cover the LSB or MSB. In both cases it is important to close the array right at the end of the last octet. This GSM Coding rule for BCD numbers prescribes that the last nibble is filled up with 1111 if the array is finished in the middle of the last octet.

	MSBit	LSBit	MSBit	LSBit
	7 8 6 5	4 3 2 1	7 8 6 5	4 3 2 1
Octett n	DIGIT_2	DIGIT_1	DIGIT_1	XXXXXXX
Octett n+1	DIGIT_4	DIGIT_3	DIGIT_3	DIGIT_2
...	: : : :	: : : :	: : : :	: : : :
Octett n+m	DIGIT_Z	DIGIT_X	DIGIT_Z	DIGIT_X

	MSBit	LSBit	MSBit	LSBit
	7 8 6 5	4 3 2 1	7 8 6 5	4 3 2 1
Octett n	DIGIT_2	DIGIT_1	DIGIT_1	XXXXXXX
Octett n+1	DIGIT_4	DIGIT_3	DIGIT_3	DIGIT_2
...	: : : :	: : : :	: : : :	: : : :
Octett n+m	1 1 1 1	DIGIT_Z	1 1 1 1	DIGIT_Z

For this reason there is two entries for BCD in the types table at the beginning of each GSM message catalogue. One for BCDODD and one for BCDEVEN. CCD uses the same function cdc_BCDODD_decode() to handle all the four cases above. It is called as cdc_BCD_decode (eRef, 1) for BCDODD and as cdc_BCD_decode (eRef, 0) BCDEVEN.

3.2.2 Non Standard Information Elements

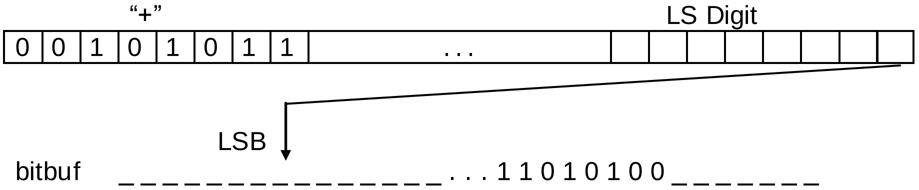
Non standard information elements are partially used in both the standard and non standard L3 messages. Some of them are older IEs and some has been added later to the protocol specifications. The coding type T30_IDENT for the fax protocol entity belongs to the second group. Rest octets and CSN.1 (compact notation described) elements has been known for a bit longer time.

3.2.2.1 T30_IDENT

The standardization document t.30 is one of the ITU Standardisation Serie T. It contains the procedures for document facsimile transmission in the general switched telephone network. One of the information fields handled in this document is the subscriber identification number. It is the international telephone number including the “+” character, the telephone country code, area code and subscriber number. This field consists of 20 octets which are to written or read as ASCII characters. This type of octet arrays are aslo used for similar information, e.g. for passwords or sub addresses.

Furthermore t.30 recommends that the information be right justified and the least significant bit of the least significant digit be the first bit transmitted. For this reason CCD reverses the information when coding/decoding of such t.30 identification numbers. For example for coding it first reverses the bytes then the bits.

Digit[20]:



Following the ITU recommendation CCD considers a “space” character for each unused octet in the information field.

3.2.2.2 CSN.1 Elements

CSN.1 elements can be subelements of standard information elements or rest octets. There are two types of CSN.1 elements. In the CCD sources they are named called CSN1_S1 and CSN1_SHL. For their simplicity we begin with the description of the first group.

3.2.2.3 CSN1_S1 Elements

Like other kinds of IE a CSN1_S1 can be made of mandatory and optional parts. The characteristic of CSN1_S1 elements relates to the behavior of their optional subelements. The presence or absence of optional subelements is given by a one-bit valid flag in the message. A valid flag of 0 means the information is not present in the IE. A valid flag of 1 is followed by the appropriate information. In other words such a subelement is made up of a fixed and an optional part. The second part exists only if the first part is filled with 1.

A simple example for a CSN1_S1 IE is „MS Radio Access Capability Value Part“ which can be depicted as follows:

```
MS RA capability value part
= {0}
```

or

```
= {1, Access Technology Type, Access capabilities, MS RA capability value part}
```

Thus this IE can have different forms. A few of them is shown in the following schema.

0

1	Access Technology Type	Access capabilities	0

1	Access Technology Type1	Access apabilities1	1	Access Technology Type2	Access capabilities2	0

1	Access Technology Type1	Access Capabilities1	1	Access Technology Type2	Access capabilities2	1	Access Technology Type3	Access capabilities1	0

When CCD is coding or decoding an IE of this category it uses the calls bf_writeBit(0), bf_writeBit(1) and bf_readBit() in order to process the valid flag.

3.2.2.4 Rest Octets

Rest octets are defined to be IEs of variable length. Thus there is no need for a length information part. Nor it is easy to find the number of octets in the coding/decoding process. CCD works with rest octets without any length information. This category of IEs do not have an identifier (IEI) either.

Different parts of a rest octet group are handled as CSN1_S1, spare padding or CSN1_SHL. The last two categories are described in the next subsections. A simple example can be NT/N rest octets. The following schemas for the bit string show there are many different possibilities for this IE.

0

1	NLP(PCH)	list of Group Call NCH information	Spare padding

where the middle part „list of Group Call NCH information“ can have one of the following forms:

0

1	Group Call Reference	0	0

1	Group Call Reference	1	Channel Descriptor	0	0

1	Group Call Reference	1	Channel Descriptor	1	0	Mobile Allocation	0

1	Group Call Reference	1	Channel Descriptor	1	1	Freq. Short List	0

In this example only CSN1_S1 and spare padding are involved.

3.2.2.4.1 Spare Padding

Both spare bits of standard IEs and spare padding in the rest octets are used to fill up an octet. Though there is a big difference between them. The padding spare is not a simple string of zeros or ones. It is related to the following schema:

0	0	1	0	1	0	1	1

While decoding a message the content of padding spare bits has no relevance, whereas a correct encoding of them is a must. In this case the output can vary depending on the length and starting position of the padding bits.
If the spare padding bits to be encoded start on an octet boundary and they are a multiple of octets, there is nothing to do but copy the above bit string to the air message.
If the padding sequence to be encoded starts in the middle of an octet, only the last bits of the above bit string must be taken in order to fill up an octet.

3.2.2.4.2 CSN1_SHL Elements

The position of the starting bit within the current octet plays a key role also for the CSN1_SHL elements. For a given position the bit is called an L bit if it has the same value as in the padding pattern above. Otherwise the bit is called an H bit.

The characteristic of CSN1_SHL elements relates to the behavior of their optional subelements. The presence or absence of optional subelements is given by a one-bit valid flag. The valid flag is an H (0 or 1) bit to signalize the presence and is or an L (0 or 1) bit to signalize the absence of the IEs.

While Coding/Decoding CCD uses the C-macro **GET_HL(bit)** to find out the H or L value for a given position. It uses the simple bit operation

```
padding_bits[bitpos%8] ^ bit
```

where the spare pattern is defined as

```
UBYTE padding_bits[8] = {0, 0, 1, 0, 1, 0, 1, 1};
```

Therefore the L value is provided by **GET_HL(0)** and the H value by **GET_HL(1).** The operation **bitpos%8** is necessary, because the global variable **bitpos** shows the position from the beginning of the message bit string and not from the beginning of the octet being processed.

3.3 Message Buffering

CCD uses two global C arrays for buffering of the message information:

```
GLOBAL UBYTE     *bitbuf, *pstruct;
```

The smaller array is bitbuf and contains the air message written in network byte order. The bigger one is pstruct into which the related message parts are written in host byte order. Both arrays are write protected through a single semaphore. In order to exchange the information with the caller function CCD uses usual pointer assignments.

While coding or decoding a message CCD processes the information given by the caller in a successive manner. While coding it reads the information successively from pstruct[], following the entries in CCD-tables[1]. The results are written into bitbuf. While decoding it reads the information successively from bitbuf[], again according to the entries in CCD-tables. The results are then written to pstruct. The bit manipulating functions of CCD are all in the file bitfun.c.

The successive processing of data is completed with an updating of written/read position after each write/read action on the information carriers bitbuf and pstruct. The used global variables for the position in bitbuf and pstruct are respectively bytepos and pstructOffs .

[1] The CCD-tables are files in the format *.cdg. They are produced by ccdgen.exe, the compiler of the message description catalogues.

In general the structural unit for several parts of an IE can be either one bit or one byte. In both cases CCD reads/writes the data byte wise from/in bitbuf or pstruct. This is very easy for the second array. For the first one CCD needs some bit operations, e.g. left or right shifting. For an instructive example suppose we want to write three bits of information in form of one byte in bitbuf. Further for simplicity suppose that the three bits should be written at the beginning of a new octet. So we need to put the three bits left adjusted in a byte and write it in bitbuf at the position bytepos. The redundant part of this byte will be overwritten in the next write action.

There are a few more global variables used by bit manipulation functions:

- **GLOBAL UBYTE bitpos**: is the number of so far written/read bits in bitbuf.
- **GLOBAL UBYTE byteoffs**: is the number of the written/read bits of the last byte in bitbuf.
- **GLOBAL USHORT bitoffs**: is the number of bits which exist in the message but are not to be processed by CCD. This part of message contains the message header information.
- **GLOBAL USHORT buflen**: is the number of the bits in the message buffer including message header information.
- **GLOBAL USHORT maxBitpos**: is the maximum value allowed for bitpos. It may be calculated or set equal to buflen.

3.3.1 Host and Network Byte Order

For writing/reading of more than one bit CCD uses a temporary variable of type t_conv16 or t_conv32. These unions are defined as below:

typedef union { UBYTE c[2]; USHORT s; } t_conv16;

typedef union { UBYTE c[4]; ULONG l; } t_conv32;

After bits are read and settled into the temporary variable they will be copied to bitbuf or pstruct. Before copying CCD may convert the byte order within this small buffer depending on the technology used. Target systems with Intel processors need such byte order conversion. The converting steps look a little different for reading out of a t_conv16 or a t_conv32. The first example shows how the bytes are new ordered while using a temporary variable of type t_conv32. The second example shows this for t_conv16.

	MSB=32			LSB=1
Intel host buffer	B0	B1	B2	B3
	Most Significant Word		Least Significant Word	

Air message in bitbuf	B3	B2	B1	B0

The second example shows how the bytes are new ordered while using a temporary variable of type t_conv16.

	MSB=32
	LSB=1	

Intel host buffer	B0	B1
Air message in bitbuf	B1	B0

3.3.2 Avoiding of overwritings

CCD avoids actively overwriting the bit buffer. This data protection happens through a logical oring or anding with an appropriate bit pattern. However the bit patterns and the logical operations are different depending on how many bits are to be written. The short descriptions below are related to the actions of coding, thus writing in bitbuf. Similar actions do exist for decoding (writing in pstruct).

1) The simple case of writing of one single bit is handled by the function bf_writeBit(). Here the constant array shift[] contains all the used bit patterns.

```
LOCAL const UBYTE shift[] = { 128, 64, 32, 16, 8, 4, 2, 1 };
```

The octet being processed is positioned by bytepos and the bit to be written is positioned with **byteoffs**.

Now shift[**byteoffs**] gives an octet filled with a 1 at **byteoffs** and zeros elsewhere. If we take the byte bitbuf[bytepos] and do a logical OR with shift[**byteoffs**] we get a 1 at **byteoffs** and the previous content elsewhere.
In order to write a 0 we take the byte bitbuf[bytepos] and do a logical AND with ~shift[**byteoffs**].

2) For writing of up to 32 bits three steps may be needed: masking, shifting and conversion of byte order. The used bit patterns for the masking are tabled in mask[]. This table contains valid masks for right-justificated values. Notice that we use a temporary variable for buffering the bits. The buffered value needs an AND operation with mask[len]. This lets the last len bits in the temporary variable be as in pstruct. All other bits will be set to 0.

At this moment the data is not ready to be copied to bitbuf. It must be adjusted in two steps. By a left shifting of the corresponding bits we guarantee that the bits will be attached to the corresponding byte just at the position **byteoffs**. The copying of data into bitbuf happens through an OR between this byte and the well adjusted bytes in the temporary buffer. By doing this we keep the previous content of the byte while attaching new bits to it.

The example case, depicted below, would be handled by the function bf_writeBits(3). 3 bits are to be written in the octet at bytepos in the position **byteoffs**(= 2). The letter x means the content is not relevant. Other letters represent some unknown content. The letters are used to emphasize the right order of bits after writing. Only two bytes of bitbuf are shown here.

Original content of bitbuf at bytepos:	a	b	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Temporary buffering: `conv.s = (USHORT) pstruct[pstructOf`	x	x	x	x	x	x	x	x	x	x	x	x	x	c	d	e
Masking: `conv.s &= (USHORT) mask[len];`	0	0	0	0	0	0	0	0	0	0	0	0	0	c	d	e
Shifting: `conv.s <<= (16 - len) - byteoffs;`	0	0	c	d	e	0	0	0	0	0	0	0	0	0	0	0
Copying into bitbuf + byte order conversion: `bitbuf[bytepos] |= conv.c[MSB_POS]` `bitbuf[bytepos + 1] |= conv.c[LSB_`	a	b	c	d	e	0	0	0	0	0	0	0	0	0	0	0

Here conv.c[MSB_POS] must refer to the first byte and conv.c[LSB_POS] to the second. For Intel processors CCD defines MSB_POS=1 and LSB_POS=0. This might be contradictory at first view. But note that referring to a char belonging to a short union works differently for different types of machins. For Intel structures conv.c[0] contains the LSB.

3.4 CCD data base

CCD works somehow as an interpreter. While coding it reads the information successively from pstruct[], following the entries in CCD-tables. The results are written into bitbuf. While decoding it reads the information successively from bitbuf[], again according to the entries in CCD-tables. The tables need to be generated by another component of the CCD-Packet called CCDGEN. CCDGEN reads the word documents containing the message and SAP description catalogues and generates files needed by CCD and test application system. The ouput (and intermediate) files are of different formats: *.h, *.cdg, *.pr, *.val (and *.mdf or *.pdf). Some of the output files are not used by CCD itself. For example header files containing the C-structure definitions are to be used by GSM applications or test programs, e.g. CCDTAP.

This chapter gives a description of the formats of the output files *.cdg. The first section however is dedicated to message description catalogues since it is instructive for understanding of the ccd tables extracted from these catalogues.

4 Message description catalogues

Currently L3 message description catalogues for GSM and GPRS are written in the common used format MS Word 97. Each file contains six chapters. The message describing part begins with the 2nd chapter where the constants used in the file are given. Type definitions in the third chapter list the coding types needed for each information element (IE). Besides the standard types like GSM4_TLV and BCDODD there are entries for special types like T30_ident and GSM5_TLV.

The 4th chapter contains tables of different L3 message structures. This shows only the outermost level. The inner part of the messages are listed with detailed information in the fifth and sixth chapter. There are two categories of IEs: basic and structured elements. Basic elements, listed in the last chapter, are simply made of variables or constants. Structured IEs are made up of some other IEs. Therefore there are links between structured IEs within the chapter 5.

For UMTS project no Winword document is used. While the standardisation uses ascii text file and ASN.1 abstract notation, the tool asn2mdf makes an MDF version of this file. Intermediate format MDF is discussed in a separate section of this document.

4.1 types

The example below is taken from the file RR.doc. The first column gives the coding type of an IE.

name	add bit	ctrl	comment
GSM1_V	0		mandatory IE V-component in one nibble
GSM1_TV	4	optional	optional with V-Component
GSM2_T	8	optional	optional IE, contains IEI only (no V-component)
GSM3_V	0		mandatory / conditional, no IEI, V-component only
GSM3_TV	8	optional	optional IE with V-component
GSM4_LV	8		mandatory / conditional, length and V-component
GSM4_TLV	16	optional	optional IE with length indicator and V-component
GSM5_V	0	optional	optional IE (bitstream to end of message)
BCDODD	4		binary coded decimal number starting with digit1
BCDEVEN	4		binary coded decimal number starting with digit2

Additional bits (**add bit**) are meant to be the T part, or the full bits eventually needed for BCD coding. This is important for ccdgen when calculating the possible bit size of IEs or messages.

Optional types are: Types requiring an identifier (T part), CSN.1, S_PADDING and GSM5_V. The latter one has been introduced for parts of 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.

Valid types for GSM and GPRS are:

GSM1_V GSM1_TV GSM2_T GSM3_V GSM3_TV GSM4_LV

GSM4_TVL GSM5_V GSM5_TLV GSM6_TLV GSM7_LV GSM1_ASN

BCDODD BCDEVEN BCD_NOFILL BCD_MNC CSN1_S1 CSN1_SHL

S_PADDING T30_IDENT

Valid types used for elements of UMTS messages are:

BITSTRING ASN1_OCTET ASN1_CHOICE ASN1_INTEGER ASN1_SEQUENCE

If you want to know which types are supported by which version of Coder/Decoder, take a look at the corresponding file ccd_codingtypes.h.

4.2 Messages

Relevant information for CCD is inserted in the tables of message description catalogues. For this reason we focus on the meaning of table entries using examples from the file RR.doc.

Note1: Absolutely important entries for coder/decoder are message ID, direction, element ID, type, ctrl and bitlen.

Note2: Indirectly or only in special cases involved entries are long name and short name.

Note3: The rest of the table entries are informative parts, yet not relevant for CCD.

4.2.1 Long name, short name, ID, direction

The first example concerns the message description for "additional assignment" in RR.doc, as depicted below. **Long name**s will be stored to mstr.cdg by CCDGEN for trace outputs of test systems. **Short name**s will be used for message structure type definitions. For d_add_assign there will be an entry **typedef struct { ... } T_D_ADD_ASSIGN**; in the file m_rr.h which is also generated by CCDGEN, see the corresponding section. The parameters **ID** and **direction**s are important to specify a message uniquely for each entity. The same PDU-Type (msg ID) can be used for both directions uplink and downlink while the message structures can be different. Valid (binary) numbers for message type are listed in 10.4 of GSM0408.

Definition:

long name	short name	ID	Direction
Additional assignment	d_add_assign	0b00111011	Downlink

Elements:

ID	long name	short name	Ref	spec ref	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

4.2.2 element ID, ref, spec ref, pres, type, len

Obviously **ID**s (also called IEI or tag) are needed to control the presence of the optional information elements. No ID is inserted in the table for a mandatory IE.

The links in the column **ref** refer to the later chapters which contain details about structured and basic elements. The links in the column **spec ref** refer to the chapters in the GSM recommendation, here GSM0408.

The columns ref, spec ref and **pres** are not used by CCD. Thus CCDGEN let them unread. The letters M, C and O stands for respectively mandatory, conditional and optional. The actual meaning of optionality for CCD is given in a previous section about coding types.

The length information in column **len** is given in bytes. Note that the length of the element "Mobile Allocation" can vary from 3 to 10 bytes. For some IEs even the length of the interval is not known since it depends on other parameters or variables. Take "BA Range" of the message "Channel Release" as an example.

0x73	BA Range	ba_range	5.1	10.5.2.1	O	TLV	6-*

The length interval for this IE is shown as "6-?" in GSM0408, 9.1.7. No upper length limit is specified except for that given by the maximum number of octets in a L3 message. In some other places the notation "-?" is used[2].

Note: Relevant length information for CCD and CCDGEN is given in descriptions of basic elements.

The column **type** gives the type of coding rules needed for the IE. For historical reasons there are some small differences between the type names declared here and the names listed in the chapter called types. For example types V and TV here are given as GSM1_V and GSM1_TV in that table. Currently valid types are listed below.

[2] Example: "facility" in the message "Alerting" in cc.doc.

In chapter Types and in *.mdf files	In column type	In ccd_codingtypes.h	Source file name
GSM1_V	V	CCDTYPE_GSM1_V	gsm_1v.c
GSM1_TV	TV	CCDTYPE_GSM1_TV	gsm1_tv.c
GSM2_T	T	CCDTYPE_GSM2_T	gsm2_t.c
GSM3_V	V3	CCDTYPE_GSM3_V	gsm3_v.c
GSM3_TV	TV3	CCDTYPE_GSM3_TV	gsm3_tv.c
GSM4_LV	LV	CCDTYPE_GSM4_LV	gsm4_lv.c
GSM4_TLV	TLV, TLV4	CCDTYPE_GSM4_TLV	gsm4_tlv.c
GSM5_V	V5	CCDTYPE_GSM5_V	gsm5_v.c
GSM5_TLV	TLV5	CCDTYPE_GSM5_TLV	gsm5_tlv.c
GSM7_LV	GSM7_LV	CCDTYPE_GSM7_LV	gsm7_lv.c
GSM6_TLV	TLV6	CCDTYPE_GSM6_TLV	gsm6_tlv.c
GSM1_ASN	ASN1	CCDTYPE_GSM1_ASN	gsm1_asn.c
BCDODD	BCDODD	CCDTYPE_BCDODD	bcdodd.c
BCDEVEN	BCDEVEN	CCDTYPE_BCDEVEN	bcdeven.c
BCD_NOFILL	BCD_NOFILL	CCDTYPE_BCD_NOFILL	bcd_nofill.c
BCD_MNC	BCD_MNC	CCDTYPE_BCD_MNC	bcd_mnc.c
CSN1_S1	CSN1_S1	CCDTYPE_CSN1_S1	csn1_s1.c
CSN1_SHL	CSN1_SHL	CCDTYPE_CSN1_SHL	csn1_sh.c
S_PADDING	S_PADDING	CCDTYPE_S_PADDING	s_padding.c
T30_IDENT	T30_IDENT	CCDTYPE_T30_IDENT	t30_ident.c
BITSTRING	Not applicable	CCDTYPE_BITSTRING	asn1_bitstr.c
ASN1_OCTET	Not applicable	CCDTYPE_ASN1_OCTET	asn1_octet.c
ASN1_INTEGER	Not applicable	CCDTYPE_ASN1_INTEGER	asn1_integ.c
ASN1_SEQUENCE	Not applicable	CCDTYPE_ASN1_SEQUENCE	asn1_seq.c
ASN1_CHOICE	Not applicable	CCDTYPE_ASN1_CHOICE	asn1_choice.c

4.3 Structured elements

Absolutely important entries for coder/decoder in the descriptions of structured elements are: **type** and **ctrl** in the **Elements** table. Indirectly or only in special cases involved entries are long name and short name. The rest of the table entries are informative parts, and not relevant for CCD.

Type means coding type. Ctrl contains additional instructions for presence conditions, array formatting and a few more special steps for special IEs, e.g. mobile identity.
The first example is the IE “Mobile Allocation” from RR.doc with the following tables:

Definition:

long name	short name	type	len
Mobile Allocation	mob_alloc	4	3-10
Mobile Allocation	mob_alloc_before	4	3-10
Mobile Allocation	mob_alloc_after	4	3-10

Elements:

long name	short name	ref	bit len	ctrl
Mobile Allocation Contents	mac	6.61	8	[1..N_MOB_ALLOC]

All the three elements mob_alloc, mob_alloc_before and mob_alloc_after has the same structure rules.
The columns **type** and **len** in the **Definition** table are not important for the interanl use of CCD. The acutal length of the structure is calculated by CCDGEN when reading chapter about basic elements. The annotation of the ctrl-column is specific for its interpreter CCDGEN. In this example the interval [1..N_MOB_ALLOC] tells us that the IE can occur in the message up to N_MOB_ALLOC times, each time of 8 bits length. The length according to the field **bitlen**. In real time CCD will read the value of this constant and set it to the maximum repeat numbers of the variable mac.

In the next example also the lower limit depends on a variable. CCDGEN will use the content of the column ctrl to insert some rules in the calc table. CCD will read this variable using the inserted rules by CCDGEN. The IEs belong to the IE “BA Range” in RR.doc.

Elements:

long name	short name	ref	bit len	ctrl
Number of Ranges	num_range		8	
Frequency Range	freq_range		20	[num_range..N_MAX_RANGE]

For BCD numbers the expression under **ctrl** helps CCD to register their coding type and the total length of the bit field. An example for this application is “ Location Area Identification” in RR.doc. The IE must be 5 bytes long but the table below gives only 5 bytes. Where is the rest of the 5 bytes? The answer is in BCD coding rules: “If the number of the digits is odd, the last Octet contains the bit pattern 1111 in the most significant nibble”. Thus for the bit field mcc 4*4 bits will be used instead of 3*4.

Definition:

long name	short name	type	len
Location Area Identification	loc_area_ident	3	5

Elements:

Long name	short name	ref	bit len	ctrl
Mobile Country Code	mcc	xxx	4	BCDEVEN[3]
Mobile Network Code	mnc	xxx	4	BCDEVEN[2]
Location Area Code	lac	xxx	16	

In the example above the bit field has a constant and known length. In some other cases the length can vary between a lower und upper limit. For the IE num of “Called party BCD number” the entry under **ctrl** is BCDEVEN[0..20] what says the variable num is a BCD number and can have up to 20*4 bits.

There is a special kind of IEs 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. Therefore they can be referred as optional elements.

The first IE of the first octet is marked with the symbol '+' in the column **ctrl**.

The last IE of the last octet is marked with the symbol '-'.

The middle IEs of between a '+' and '-' are not marked with any symbols.

A single octet of this type is marked with '*'.

The example below belongs to the IE "Cause" from cc.doc. From cs till cause there can be up to 3 octets in this group.

Definition:

long name	**short name**	**type**	**len**
Cause	cc_cause	4	4-32
Cause	cc_cause_2	4	4-32

Elements:

long name	**short name**	**Ref**	**bit len**	**ctrl**
Coding standard II	cs	6.6	2	+
Spare	.0		1	
Location	loc	6.25	4	-
Recommendation	rec	6.46	7	*
Cause value	cause	6.3	7	*
Diagnostics	diag	6.11	8	[0..MAX_DIAG_LEN]

Any further octets relating to a protocol extension are marked with '!' or '#'. The last sign marks the end of such an extended group. For the example below this means:
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.

Definition:

long name	short name	ID	direction
BCS Digital transmit command	BCS_DTC	0b10000001	both

Elements:

long name	short name	ref	ref	pres	len	ctrl
Facsimile control field	fcf		5.3.6.1	M	1	
basic facsimile capabilities receiver	cap0_rcv		5.3.6.2.1	M	3	
extended facsimile capabilities 1 receiver	cap1_rcv		5.3.6.2.1	O	1	!
extended facsimile capabilities 2 receiver	cap2_rcv		5.3.6.2.1	O	1	!
extended facsimile capabilities 3 receiver	cap3_rcv		5.3.6.2.1	O	1	!
extended facsimile capabilities 4 receiver	cap4_rcv		5.3.6.2.1	O	1	!
extended facsimile capabilities 5 receiver	cap5_rcv		5.3.6.2.1	O	1	!
extended facsimile capabilities 6 receiver	cap6_rcv		5.3.6.2.1	O	1	!
extended facsimile capabilities 7 receiver	cap7_rcv		5.3.6.2.1	O	1	#

Up to here we have learnt the notation used in ctrl column to describe IEs of an extended group or fields of repetitive elements. The second category is a common source for misunderstandings and failure. Hence we dedicate a table to sum up the discussed alternatives. The following table gives a summary of how bit or IE fields are introduced in message description (*.doc) files.

Field Type	**Msg Desc File**	**IE short name**	**Ctrl-column**	**Declarations in Header file *.h**	**Bitsize of each field element**	**repType in melem.cdg**
Constant length; Field made of a simple IE	gmm.doc	rand_value	[16]	UBYTE rand_value[16];	8	'c'
Constant length; Field made of a structured IEs	rr.doc	tagged_usf_tn	[8]	T_tagged_usf_tn tagged_usf_tn[8];	4 or 0	'c'

Constant length; Bit field	gmm.doc	tmsi	[.32]	BUF_tmsi tmsi; ....
				typedef struct {
				USHORT l_tmsi;
				USHORT o_tmsi;
				UBYTE b_tmsi[5];
				} BUF_tmsi;
					1	'b'

Variable lenght; Bit field	Warp\ rr.doc	allo_bmp7	[.allo_len7..127]	BUF_allo_bmp7 allo_bmp7;
				...
				typedef struct
				{
				USHORT l_allo_bmp7;
				USHORT o_allo_bmp7;
				UBYTE b_allo_bmp7[20];
				} BUF_allo_bmp7;
					1	'b'

Variable lenght saved to a counter; Field made of simple IEs	Gsm\ cc.doc	subaddr	[0..20]	UBYTE c_subaddr;
				UBYTE subaddr[20];
					8	'i'

Variable lenght saved to a counter; Field made of structured IEs	Gsm\ rr.doc	gr_call_info	[0..MAX_GR_C_INFO]	typedef struct {
				...
				UBYTE c_gr_call_info;
				T_gr_call_info gr_call_info[5];
				} T_nt_rest_oct;
					unknown	'i'

Variable length must be calculated by ccd;	Gprs\ grr.doc	inst_bitmap	[psix_cnt+1..16]	UBYTE c_inst_bitmap;
				UBYTE inst_bitmap[16];
					1	'v'

Field made of simple IEs						

Variable lenght must be calculated by ccd; Field made of structured IEs	Gsm\ rr.doc	cod_prop	[cnt_cod_prop..3]	UBYTE c_cod_prop;
				T_cod_prop cod_prop[3];
					10	'v'

Variable lenght must be calculated by ccd; Field made of simple IEs	Gprs\ sms.doc	num	BCDEVEN[0..20]	UBYTE c_num;
				UBYTE num[20];
					4	'v'

Another type of control information in the field **ctrl** relates to the conditional expressions. The table below describes the structure of the IE “Channel Description” from RR.doc. The presence of the IEs arfcn, maio and hsn depends on the value of the basic IE hop. If this variable is 0 the 12 bits after it are made of an spare field (00) and 10 bits for the variable arfcn. However if this variable is 1 the 12 bits after it are made of 6 bits dedicated to maio and 6 bits to hsn.

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 most sophisticated rules in the column ctrl are dedicated to a special IE called “Mobile Identity”.

Definition:

long name	**short name**	**type**	**len**
Mobile Identity	mob_ident	4	3-10

Elements:

long name	**short name**	**ref**	**bit len**	**ctrl**
Type of identity	ident_type		3	(GETPOS,:,4,+,:,1,+,SETPOS)
Odd/ Even indicaction	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]

In the following table you find a summary of notations in the ctrl column and how they are understood by CCD.

Instruction	**meaning of instruction**	**example**
[0..CONSTANT]	array of bytes (also USHORT in *.pdf)	[0..MAX_RFL_NUM_LIST]
[var-name+number..CONSTANT]	array of bytes (also USHORT in *.pdf)	[rfl_cont_len+3..19]

[.CONSTANT]	array , dot marks a bit array	[.32]
BCDODD[numbers]	BCD numbers starting with digit1	
BCDEVEN[numbers]	BCD numbers starting with digit2	BCDEVEN[2] or BCDEVEN[0..20]
{ … }	conditional	{flag=1 AND flag2=1 OR flag=0}
(…)	command sequence	(GETPOS,:,4,+,:,1,+,SETPOS)
GETPOS	get the bitstream pointer	(GETPOS,:,4,+,:,1,+,SETPOS)
SETPOS	set bit stream pointer	As above
KEEP,regNr	keep value of a variable in ccd register	(KEEP,1) see GRR.doc chapter 5.65
TAKE,regNr	Take the value of ccd register	[.(TAKE,1)+1..8] see GRR.doc chapter 5.136
MAX,regNr	Compare and keep the maximum in ccd register from a variable and ccd register	(MAX,2) see GRR.doc chapter 5.73 and 5.74
:	duplicate the element	(GETPOS,:,4,+,:,1,+,SETPOS)
^	swap the two elements	see CC.doc chapter 5.4 bearer capability
+ * -	first middle last octett	see CC.doc chapter 5.4 bearer capability
AND OR XOR	logical operations: AND, OR and XOR	{flag=1 AND flag2=1 OR flag=0}
= # < >	comparisons	(KEEP,1) {n_r_cells # 0}
(22) or (0)	Maximum length of spare padding bits	S_PADDING .00101011 (22) means if the message consists of less than 22 bytes then fill up with the bit pattern S_PADDING .00101011 (0) means if the message doesn't end on octet boundary then fill up to octet boundary with bit pattern eg.: xxxx1011

4.4 Basic elements

The tables in the last chapter of a message catalogue contains values or value ranges for the so called basic elements. A basic IE is simply made of one variable. Single values will be C-macros in the source code. For each entity CCDGEN puts all such C-macro definitions in a file with the extension *.val, e.g. in m_rr.val for RR.doc.

Furthermore CCDGEN produces the tables mconst.cdg and pconst.cdg which contain the values or value ranges of the basic IE. The Format of the table will be discussed in a later section of this document.

The two example tables, below, for single values and value ranges do not seem to need a further description.

Definition:

long name	short name	bit len
A5/2	a5_2	1

Values:

value	c-macro	Comment
0	A5_2_NO	encryption algorithm A5/2 available
1	A5_2_YES	encryption algorithm A5/2 not available

Definition:

long name	short name	bit len
Lowest ARFCN	low_arfcn	7

Values:

value	comment
1-124	
DEF	reserved

5 CCDGEN Intermediate Files *.mdf and *.pdf

Before CCDGEN produces its final output files, another tool (xgen.exe or asn1_to_mdf.exe) must have processed the information in the message and SAP catalogues. Such a tool puts the relevant information in the so-called intermediate files with the extension *.mdf and *.pdf. A comparison between the content of these files and the *.doc files shows that some information in the *.doc files are redundant and thus not necessary for the CCD tables.
The different categories of information in these files are labeled with CONST, TYPE, VAR, VAL, COMP, UNION, MSG (or PRIM), all occurring in the same order. Each category has its own format. In the examples below the format of each group is given on the top the single entry. The examples belong to the entity RR.

1) CONST

Key Word	Name	Value	Comment
CONST	L3MAX_ACK	251	; GSM 4.06, section 5.8.5

2) TYPE

Key Word	Coding Type	addbits	Comment
TYPE	GSM1_V	0	; mandatory IE V-component in one nibble

3) VAR

Key Word	Short Name	Long Name	Bit Lenght
VAR	access_ident	"Access identity"	2

4) VAL

Key Word	Value	Short Name	Long Name (or empty)
VAL	0	AI_OCT_ID	"octet identifier"
VAL	DEF	"reserved"	

4) COMP
Structured IEs are described in *.mdf under the label COMP (composition). The important parameters for each composition are name, ID and the entries in the column **ctrl**. The missing parameter Coding Type is given under the label MSG together with the name of the IE. Also the parameter Bit Length (in the column **bitlen)** is not really forgotten. It is given under the label VAR.

The format given below is valid for all IEs but the sophisticated IE mob_ident.

```
Key Word        Short Name          Long Name     ID
{
BCD Type or empty     (Short Name) or (Short Name + ctrl column)     Comment
}
```

```
COMP        ba_range          "BA Range"  0x73
{
       num_range                                       ;  Number of Ranges
       freq_range [num_range..N_MAX_RANGE]             ;  Frequency Range
}

COMP  loc_area_ident      "Location Area Identification"
{
    BCDEVEN              mcc [3]                       ;  Mobile Country Code
    BCDEVEN              mnc [2]                       ;  Mobile Network Code
                         lac                           ;  Location Area Code
}
```

5) MSG
This category of information belongs only to the message description files *.mdf. The used structure is a unifying of the two tables “description” and “elements” for each message in *.doc. Each message is

uniquely defined by its ID and direction. The information elements of a message are to find under the label COMP.

```
Key Word    Short Name          Direction    ID           Comment
{
            Coding Type  Short Name          Comment
}

MSG         b_rr_status         both         0b00010010  ;  RR Status
{
            GSM3_V      msg_type             ;  Message Type
            GSM3_V      rr_cause             ;  RR Cause
}
```

6) PRIM

This category of information occurs only in the SAP description files *.pdf.

```
 Key Word   C-macro                     ID
{
            Short Name          Comment
}
PRIM        MMCC_RELEASE_REQ  0x0202
{
            inst_id            ;  Instance number
            ti                 ;  transaction identifier
}
```

6 CCDDATA or CCD tables

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.

6.1.1.1 ccdmtab.cdg and ccdptab.cdg

This file contains 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"
};
```

6.1.1.2 mstr.cdg

Although this table is not used by CCD 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",
```

6.1.1.3 mconst.cdg

A big part of these files contains definitions of c-macros for IDs, bit lengths etc for all entities (or SAPs). The small rest 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 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   */
...
```

6.1.1.4 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 are 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.

6.1.1.5 mval.cdg

Although this table is not used by CCD 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 are 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.

Value ranges are given by the first (**startVal**) and last (**endVal**) value number. For single values the **startVal** is equal to **endVal**.

6.1.1.6 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 are 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.

6.1.1.7 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 is dependent 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 four lines.

```
/*  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 indicaction	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 an 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 },
```

long name	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	ASN 1	6.42		3
0x04	Unstructured SS data string	ussdString	ASN 1	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 and C which abbreviate respectively variable, spare and composition. Depending on the element type the parameter **ref** must be interpreted as an index referring to the entry in mvar, spare or mcomp table.

6.1.1.8 mcomp.cdg

This table contains a few elementary parameters to specify an IE. The format of the entries are 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.

6.1.1.9 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 are 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]);
```

6.1.1.10 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 bitstream 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

The UPN calculator is used by CCD when it checks a condition or when it reads the value of a variable or constant.

6.1.2 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 enty 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 (condCalc-Ref=57) from the table calc in order to check the codition 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 an TLV type IE. The T part is embedded in the entry in melem:

```
/*  971*/ {   7, 1, ' ', ' ',    0,     0,    19, 0x72  , 'C',    173 },
```

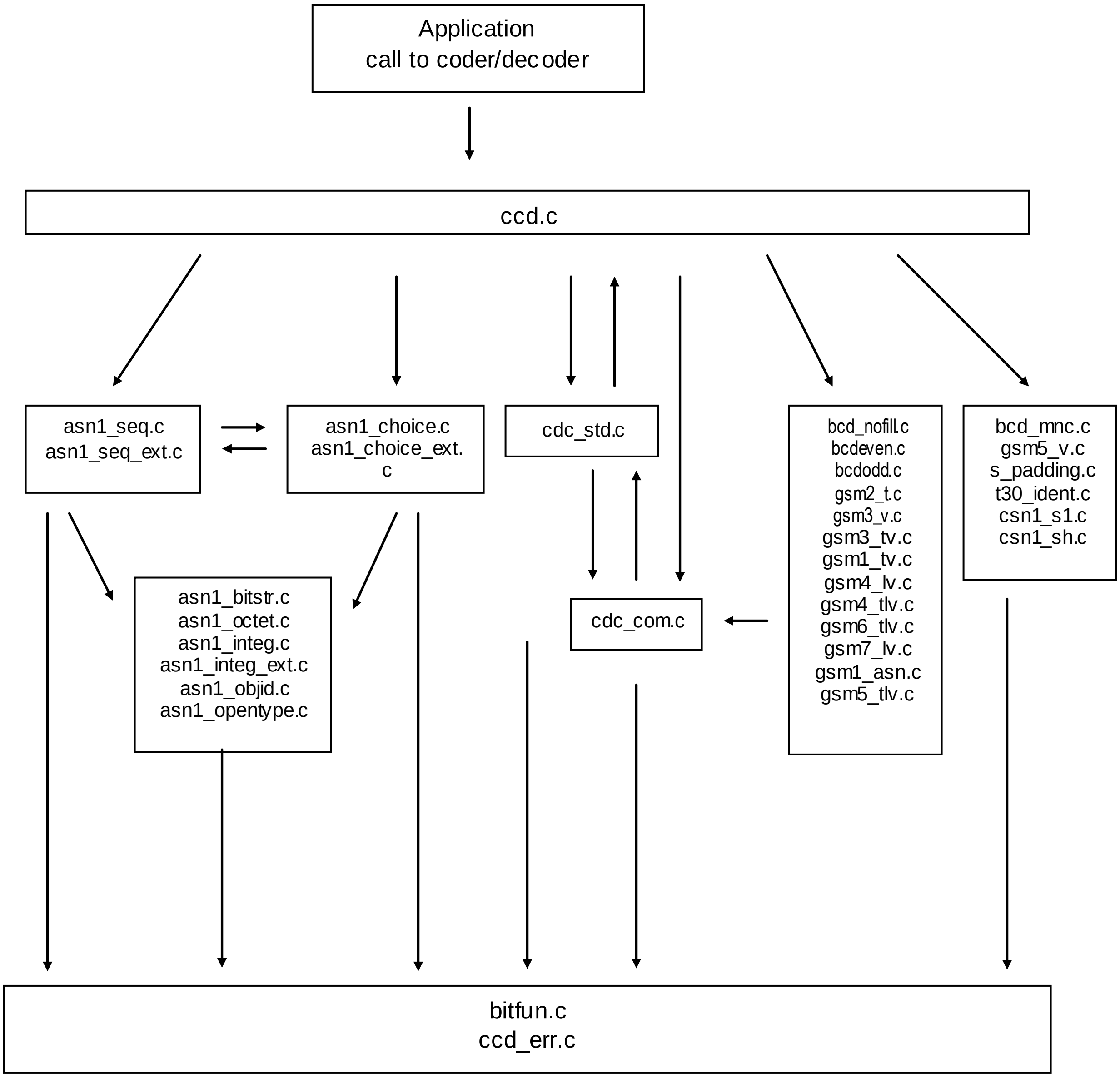
and L part is a dynamic value which is calculated and written in the message according to each message structure.

7 CCD Modules

The main module of CCD is **ccd.c**. It contains the entry functions like ccd_encodeMsg() and ccd_decodeMsg() which initiate encoding and decoding of different sorts of information elements. Besides these CCD-Application interface (see also ccdapi.h) offers a few other functions, e.g. ccd_codeByte in **cdc_std.c**, for the use of entities to do simple encoding/decoding procedures. CCD also offers functions like ccd_getFirstError() and ccd_getNextError() (in **ccd_err.c)** to the applications so they can read the CCD internal error code whenever errors have occurred.

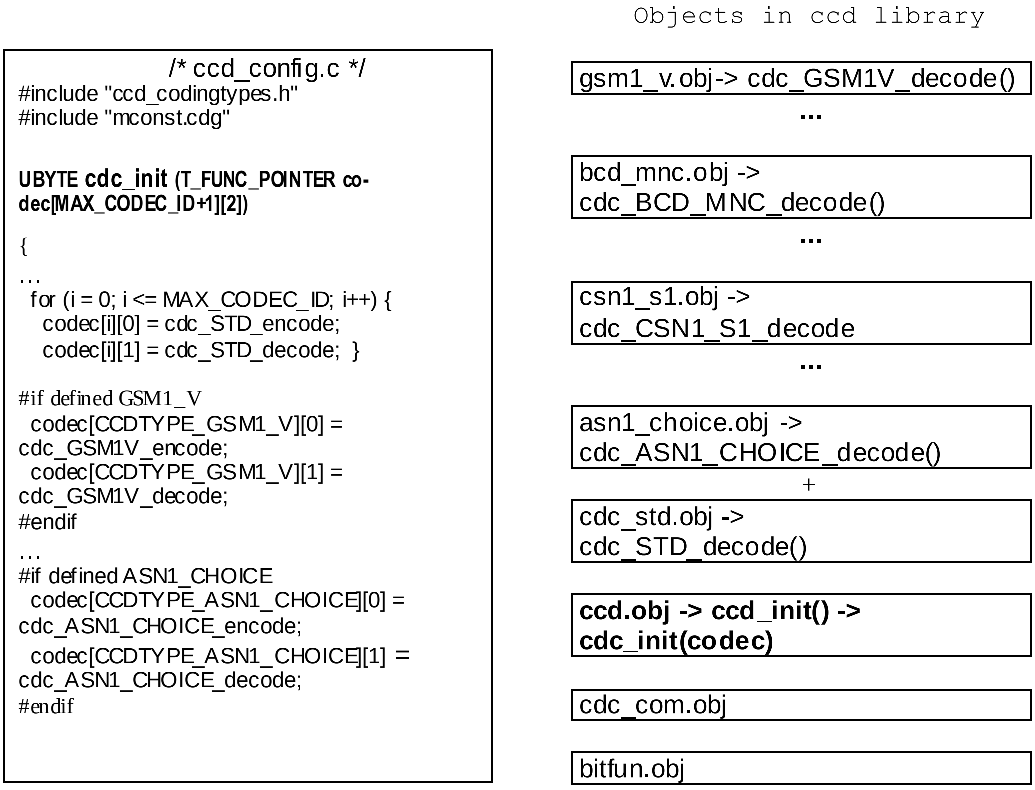
The figure below shows the hierarchy of function calls in CCD modules. It is valid only for the basic coding/decoding procedures triggered by calls to ccd_codeMsg, ccd_decodeMsg, ccd_codeMsgPtr and ccd_decodeMsgPtr. The bit manipulating functions (in **bitfun.c**) is invoked by most of the modules.

The module cdc_com.c contains common used functions of all other modules. The module cdc_std.c contains functions for code/decode of standard or simple elements. That means elements without special coding rules.



As can be seen from the schema above, there are several source code modules for several coding types. Indeed a CCD library contains all functions ever implemented for CCD. These types are listed in **ccd_codingtypes.h**. In general each application, as a user of CCD, needs only a subset of all these coding functions. This means for CCDDATA a list of the involved coding types for a given product, e.g. listed in mconst.cdg which is generated by CCDGEN.

However because of the strict separation of CCD and CCDDATA, the only chance to transfer this information from CCDDATA to CCD is to configure a table at the link time and pass it to CCD at the run time. The figure below describes the configuration behaviour. Note that ccd_config.c is not a module of CCD but CCDDATA.



```
                  /* ccd_config.c */
#include "ccd_codingtypes.h"
#include "mconst.cdg"

UBYTE cdc_init (T_FUNC_POINTER co-
dec[MAX_CODEC_ID+1][2])

{
…
  for (i = 0; i <= MAX_CODEC_ID; i++) {
    codec[i][0] = cdc_STD_encode;
    codec[i][1] = cdc_STD_decode;  }

#if defined GSM1_V
  codec[CCDTYPE_GSM1_V][0] =
cdc_GSM1V_encode;
  codec[CCDTYPE_GSM1_V][1] =
cdc_GSM1V_decode;
#endif
…
#if defined ASN1_CHOICE
  codec[CCDTYPE_ASN1_CHOICE][0] =
cdc_ASN1_CHOICE_encode;
  codec[CCDTYPE_ASN1_CHOICE][1] =
cdc_ASN1_CHOICE_decode;
#endif
```

Objects in ccd library
gsm1_v.obj-> cdc_GSM1V_decode()
...
bcd_mnc.obj -> cdc_BCD_MNC_decode()
...
csn1_s1.obj -> cdc_CSN1_S1_decode
...
asn1_choice.obj -> cdc_ASN1_CHOICE_decode()
+
cdc_std.obj -> cdc_STD_decode()
ccd.obj -> ccd_init() -> cdc_init(codec)
cdc_com.obj
bitfun.obj

7.1 Global macros, prototypes, types and variables

7.1.1 Global C-macros

C Preprocessor - macros	Comment (For what reason, the macro is set?)
__cplusplus	For compatibility
_MSDOS	To include dos.h and conio.h
_TMS470	CCD will be compiled for a target board.
WIN32	CCD will be compiled for a win32 target.
BUFFER_ALIGNMENT	Size value for forcing n-byte aligned addresses to byte arrays
CCD_C CCD_ERR_C CDC_STD_C	To protect against multiple function prototype definition when including ccdapi.h.
CCD_CDGINDEP	Definition of T_MSGBUF independent of MAX_BITSTREAM_LEN which is given by CCDDATA.
CCD_GPRS_ONLY	Allocate a further global structure for GRR.
CCD_MK	
CCD_SYMBOLS	Traces will contain the name of elements (from mcomp and mvar in CCDDATA).
CCD_TEST	CCD will be linked to a one-task system and without any frame (ccdtest.exe)
DEBUG_CCD	Trace output for the caller entity, where the trace class TC_CCD is switched on.
DYNAMIC_ARRAYS	Dynamic memory allocation for elements of pointer type.
ERR_TRC_STK_CCD	To save CCD_ID (element reference number) in a cascade form for error reporting with more details.
FAR_MODEL	System will use far model jump table.
M_INTEL (M_MOTOROLA)	For using Intel (Motorola) type processor on the target
NEW_FRAME	Objects need functions from the new version of frame.
OLD_FRAME_TAP	Objects will be linked to TAP with an old version of frame.
SHARED_CCD	CCD will be linked to a preemptive multithreaded system (uses old or new frame)
SHARED_VSI	CCD will be linked to a preemptive multithreaded system (uses old or new frame)
USE_DRIVER	CCD is going to be linked, it will not be used as a driver.

7.1.2 Prototypes

```
GLOBAL SHORT (*codec[MAX_CODEC_ID][2])(USHORT, USHORT);
```

7.1.3 Global types

The table below contains all the global types which are described in the next subsections.

Name	Description

T_CCD_CompTabEntry	contains parameters for compositions (structured IEs)
T_CCD_ElemTabEntry	contains parameters for all IEs
T_CCD_VarTabEntry	contains parameters for IEs which are variables
T_CCD_ValTabEntry	contains parameters for values of variables
T_CCD_SpareTabEntry	contains parameters for IEs which are spare bits
T_CCD_CalcTabEntry	contains parameters for calculation steps for a UPN calculator
t_conv16	contains types to protect the byte order conversion
t_conv32	contains types to protect the byte order conversion

7.1.3.1 T_CCD_CompTabEntry – Definition entry for a composition

Definition:

```
typedef struct
{
#ifdef CCD_SYMBOLS
  char    *name;
  USHORT  longNameRef;
#endif
  USHORT  cSize;
  USHORT  bSize;
  USHORT  numOfComponents;
  USHORT  componentRef;
} T_CCD_CompTabEntry;
```

Description:

T_CCD_CompTabEntry contains specific parameters of a structured information element. The corresponding CCD table is mcomp.cdg which contains values for the parameters of all possible IEs according to the message description catalogues. The number of bits used for an IE is stored to bSize. The space needed in the C-structure is given by cSize. The member numOfComponents gives the maximum number of IEs that the structured IE can contain. The member componentRef is an index referring to the entry in the table “melem”.

The members *name and longNameRef are respectively the short and long names of the IE given in the corresponding message description catalogue.

7.1.3.2 T_CCD_ElemTabEntry - Table entry for an element

Definition:

```
typedef struct
{
  UBYTE   codingType;
  BOOL    optional;
  char    extGroup;
  char    repType;
  USHORT  maxRepeat;
  USHORT  structOffs;
  USHORT  ident;
  char    elemType;
  USHORT  elemRef;
} T_CCD_ElemTabEntry;
```

Description:

T_CCD_ElemTabEntry contains specific parameters needed for composing an IE. The corresponding CCD table is melem.cdg which is generated by CCDGEN.

The member codingType is necessary to choose the appropriate CCD encoding/decoding functions. Valid types are listed in the file mconst.cdg.

Optional IEs are marked by the member optional. For some optional IEs the condition to be checked is expressed in the table calc.

For extended groups the member extGroup shows whether the IE appears in the beginning, in the middle or at the end of the extended group. This member is 0 if the IE does not belong to an extended group.

For IEs made of repeating variables the member repType is one of the characters i, b, c and v which stand for respectively interval, bit field, constant and variable. If the message description gives a maximum number for the repeats the structure member maxRepeat is different from 0.

If the IE belongs to a structured IE the member structOffs is different from 0 and refers to its place within the whole composition. If the IE has an information element identifier the member ident is different from 0 and contains the identifying number.

The member elemType can be one of the characters V, S and C which mean respectively variable, spare and composition. Depending on the element type elemRef must be interpreted as an index referring to the entry in “mvar”, spare or mcomp table.

7.1.3.3 T_CCD_VarTabEntry – Table entry for a variable

Definition:

```
typedef struct
{
#ifdef CCD_SYMBOLS
  char     *name;
  USHORT   longNameRef;
#endif
  USHORT   bSize;
  USHORT   cSize;
  char     cType;
  UBYTE    numValueDefs;
  USHORT   valueDefs;
} T_CCD_VarTabEntry;
```

Description:

T_CCD_VarTabEntry contains specific parameters of an information element. The corresponding CCD table is mvar.cdg which contains all the needed variables according to the message description catalogues. The number of bits used for a specific variable is stored to bSize. The space needed in the C-structure is given by cSize and cType. The member numValueDefs gives the maximum number of possible values for a variable. The member valueDefs is an index referring to the entry for the first value in the table “mval”.

The members *name and longNameRef are respectively the short and long names of the IE given in the corresponding message description catalogue.

7.1.3.4 T_CCD_ValTabEntry – Table entry for values of variables

Definition:

```
typedef struct
{
  USHORT   valStringRef;
  BOOL     isDefault;
  ULONG    startValue;
  ULONG    endValue;
} T_CCD_ValTabEntry;
```

Description:

T_CCD_ValTabEntry contains parameters to specify a single value or value range. For single values the member startValue and endValue are identical. For values which are a default for a variable the member isDefault should be 1. The corresponding CCD table for this structure is mval which contains values for all possible variables according to the message description catalogues. The index valStringRef refers to the entry in mstring table for the descriptive string about the specific value.

7.1.3.5 T_CCD_SpareTabEntry – Table entry for spare bits

Definition:

```
typedef struct
{
  ULONG    value;
  UBYTE    bSize;
} T_CCD_SpareTabEntry;
```

Description:

T_CCD_SpareTabEntry contains parameters for composing a group of spare bits. The corresponding CCD table is spare which contains similar parameters for all possible spare bit groups according to the message description catalogues. The number of spare bits is stored to bSize. The member value gives the hexadecimal form of the value shown by the spare bits.

7.1.3.6 T_CCD_CalcTabEntry – Definition entry for a calculation

Definition:

```
typedef struct
{
  char     operation;
  ULONG    operand;
} T_CCD_CalcTabEntry;
```

Description:

T_CCD_CalcTabEntry contains parameters to specify the calculation steps for the UPN calculator. The corresponding CCD table is calc. Each entry of the table represents one step. The meaning of the member operand depends on the character shown by the member operation, see also the section about calc.cdg.

7.1.3.7 t_conv16 - Conversion structure

Definition:

```
typedef union
{
  UBYTE c[2];                /* 2 bytes <-> USHORT */
  USHORT s;
} t_conv16;
```

Description:

This type is used to protect the byte order conversions while reading/writing of 16 bits values.

7.1.3.8 t_conv32 - Conversion structure

Definition:

```
typedef union
{
  UBYTE c[4];
  ULONG l;
} t_conv32;
```

This type is used to protect the byte order conversions while reading/writing of 32 bits values.

7.1.4 Global variables

`UBYTE  *bitbuf`	points to the array containing the encoded message.
`UBYTE  *pstruct`	points to the array containing the decoded message.
`USHORT pstructOffs`	holds the byte position in *pstruct; refers to the variable that is currently in process.
`USHORT bytepos`	holds the position in bitbuf; refers to the byte that is currently in process.
`USHORT bitpos`	is the number of so far written/read bits in bitbuf.
`UBYTE  byteoffs`	is the number of the written/read bits in the lastly processed byte of bitbuf.
`USHORT bitoffs`	is the number of bits which exist in the message but are not to be processed by CCD. This part of message contains the message header information.
`USHORT buflen`	is the number of the bits in the message buffer including the message header information.
`USHORT maxBitpos`	is the maximum value allowed for bitpos. It may be calculated or set to buflen.
`USHORT lastbytepos16`	
`USHORT lastbytepos32`	
`UBYTE  ccd_recurs_level`	is the level of the IE that is currently in process within a nested IE.
`BOOL   TagPending`	is the flag for reading the T value of an standard IE.
`T_CCD_VarTabEntry mvar[]`	Is the table containing parameters that describe variables of all IEs.
`T_CCD_SpareTabEntry spare[]`	Is the table containing parameters that describe spare bit fields.
`T_CCD_CalcTabEntry  calc[]`	Is the table containing UPN operations needed to compose some IEs.
`T_CCD_CompTabEntry  mcomp[]`	Is the table containing parameters that describe all IEs.
`T_CCD_ElemTabEntry  melem[]`	Is the table containing parameters that describe the different elements of all IEs.

7.2 ccd.c

7.2.1 Includes, macros, types and variables

7.2.1.1 Includes

Besides the header files of the C standard library the following header files are required:

- In **typedefs.h** (this has substituted the former file **stddefs.h**) global types like LONG, ULONG etc. are defined.
- The prototypes of bit manipulating functions of bitfun.c are collected in **bitfun.h**.
- The types used for the variable fields called CCD tables are defined in **ccdtable.h**.
- The file **ccd.h** contains prototypes of the local functions of the module ccd.c and a few global macros like BREAK and CONTINUE.
- The file **ccdapi.h** is an interface to applications. It contains prototypes of the global functions of the module ccd.c and a number of global macros. These prototypes and macros will be shared be-

tween CCD and the application using it. A good example for the macros is the list of the CCD error codes or the return value ccdOK.

- All custom specific definitions are collected in **custom.h**.
- All definitions of global constants, types, and macros for the GSM Protocol Stack are collected in **gsm.h**.
- The file **vsi.h** contains definitions for the virtual system interface (VSI). One of the services of this interface is creating, obtaining, releasing and deleting of semaphores. CCD needs semaphores to protect its shared message buffer.
- As long as the CCD data tables are not loaded dynamically in the run time we need to include the file **ccdmtab.cdg**. The constructions in this file will define variables which will contain the data read by CCDGEN from the message catalogue word documents.
- For the dos version of CCD which is no more of interest the header files **dos.h** and **conio.h** must be included.

7.2.1.2 Macros

VSI_CALLER	is the task handle of CCD when calling to VSI functions
MAX_CODEC_ID	is the maximum number of encoding/decoding functions called via codec[][].
ENCODE_FUN (DECODE_FUN)	is used to distinguish between encoding and decoding function when calling to codec[][].
MAX_UPN_STACK_SIZE	is the size of UPN-Stack
MAX_ERRORS	is the maximum number of errors in the CCD internal error list

7.2.1.3 Types

Definition:

```
typedef struct
{
  UBYTE  error;
  UBYTE  numErrPar;
  USHORT errPar    [MAX_ERR_PAR];
} T_CCD_ERR_ENTRY;
```

Description:

T_CCD_ERR_ENTRY contains parameters to manage the list of detected errors by CDD while coding/decoding. CCD makes a diagnosis about the detected errors and writes about them in an error list of the type T_CCD_ERR_ENTRY. The registered errors can be queried by the entities if necessary or interesting for them.

The first member of the struct **error** is unique for each diagnosis. Valid error numbers are defined in the CCD-Application interface ccdapi.h. Often some parameters are attached to an error code so that the application can also do its own diagnosis based on the reported parameters by CCD. The number of parameters related to each error report is stored to the structure member **numErrPar**. The value of the parameters themselves are put to the member **errPar**. Each entity can have access to the list through the functions ccd_getFirstError() and ccd_getNextError(). Though the list is an internal resource of CCD and the struct type T_CCD_ERR_ENTRY is defined only for a local scope.

7.2.1.4 Variables

The error handling by CCD is not restricted to a function return value stored in `CCD_Error`. It also puts its diagnosis about the detected errors in the list **errlist** which contains up to MAX_ERRORS errors for each entity. The number of reported errors for each entity is registered in **numCCDErrors**. The entity that has called CCD for coding or decoding is registered in the variable **aktEntity** for a later use in addressing to errlist.

```
SHORT           CCD_Error;
T_CCD_ERR_ENTRY       errlist[NUM_OF_ENTITIES][MAX_ERRORS];
UBYTE            numCCDErrors[NUM_OF_ENTITIES];
UBYTE            Errnum[NUM_OF_ENTITIES];
UBYTE           aktEntity;
```

After an error detection by CCD the function ccd_setError() is called. In some scenarios it may interrupt the normal operation of CCD through a long jump to the return line of the function ccd_encodeMsg or ccd_decodeMsg. The parameter needed for calling to setjmp() and longjmp() is **jmp_mark** . Note that the same code line can be reached either through a normal operation or a long jump. The flag **jmp_mark_set** will be TRUE if the code line is reached by a long jump.

```
jmp_buf jmp_mark;
BOOL    jmp_mark_set = FALSE;
```

The variable **mi_length** is the length of the Id (or IEI) of the first information element in the message. It is given to CCD by the entity. CCD reads/writes from/to the array bitbuf so many bits as mi_length.

```
UBYTE mi_length[NUM_OF_ENTITIES];
```

The initialisation flag **initialized** is to prevent multiple calls to the function ccd_init() by the entities using CCD. It is set to TRUE after the initialisation.

```
BOOL initialized = FALSE;
```

To realize the calculation steps registered in the CCD table calc a UPN calculator is implemented in the CCD code. The buffer necessary for such operations is **stack** which will be over flown if the variable **StackOvfl** is set to TRUE. The index **SP** helps to refer to a special element on the stack.

```
LOCAL UBYTE SP;
LOCAL ULONG Stack[MAX_UPN_STACK_SIZE];
LOCAL BOOL  StackOvfl = FALSE;
```

In a preemptive multithreaded system each entity is able to call the functions ccd_decodeMsg() or ccd_decodeMsg and manipulate the coding and decoding section. It is necessary to protect the critical sections like bitbuf and pstruct with the help of semaphores. The semaphores are created in the function ccd_init() which must be called before any other call to the CCD functions. Each call to ccd_codeMsg() or ccd_decodeMsg() leads to an attempt to obtain the semaphore. The semaphore handles are stored to the variables **semCCD_Codec** and **semCCD_Buffer**.

```
#ifdef SHARED_CCD
  T_VSI_SHANDLE semCCD_Codec, semCCD_Buffer;
```

The semaphore with the handle CCD_Buffer is planned to protect the internal CCD Buffer named **decMsgBuffer**.

```
LOCAL UBYTE decMsgBuffer[MAX_MSTRUCT_LEN];
```

This buffer has a special usage and is used whenever a null pointer is given to ccd_codeMsg() or ccd_decodeMsg(). CCD uses this buffer for writing the encoded or decoded data into it. The application using this facility needs to call ccd_begin() and ccd_end() before entering the critical section through a call to ccd_codeMsg() or ccd_decodeMsg(). Using an internal buffer for CCD helps to save buffer space but makes entities slower.

If the switch DEBUG_CCD is on, it will be possible to watch the progress of CCD operation. In case of ARM7 processors a further variable is used to signalize special case. The special case is described by messageId=0xfe. The actual meaning of this condition is that no trace reports should be written to the output file.

```
#ifdef DEBUG_CCD
  #if defined (_TMS470)
    LOCAL BOOL TraceIt=FALSE;
```

7.2.2 ccd_init()

ccd_init() initializes CCD und sets the flag **initialized**. It is one of the interface functions of CCD to applications. It must be called by the application before any other calls to CCD functions.

- The two VSI calls to vsi_s_open create two semaphores and write their handles in the variables **semCCD_Codec** and **semCCD_Buffer.** The semaphores are to protect the common buffers bit-buf and decMsgBuffer.
- We will see later that the first action done for encoding/decoding is writing/reading of the message identifier. For this action CCD needs to know how many bits in the bit stream present the message Id. There is an information element called msgType in the table melem from which CCD can read the bit size of the msgId. The next step CCD does in ccd_init() is to find the place of msgType in melem: First it looks in the table of all valid messages (mmtx) for the place of the first valid message in mcomp. With the reference number read in mcomp then it looks for the first element of the message which is msgType typically. If this is successful CCD reads the place of the variable msgType in the table mvar. There the bit size is ready to be read.
- In the next step ccd_init() initialises an internal table named jump table. This table will contain the addresses of the registered functions for encoding and decoding of the elements of all the used coding types. After initialisation of the jump table ccd_init() registers the used functions. Such a registration is necessary since different GSM applications need different coding types although they have a big part in common.

7.2.3 ccd_register

This function provides a different Coder/Decoder initialization. If the calling entity shall use CCD in a reentrant manner (i.e. without being synchronized with other entities when using CCD) ccd_register(CCD_REENTRANT) must be called instead of ccd_init().

It must be guaranteed that enough memory in the D-partitions is available in the order that CCD can allocate a set of local data for each task which must use CCD in a reentrant manner. The batch file \gpf\CCD\util\globs.bat can be executed to print the size of memory needed for one task.

For that purpose, the constant in xxxconst.h, denoting the number of dynamic partitions for the CCD data sets this is currently #define DMEMPOOL_1_PARTITIONS 1 must match the number of tasks calling ccd_register.

7.2.4 ccd_codeMsg()

ccd_codeMsg() is another function of the CCD interface to applications. It is called by an application to trigger an encoding action. The parameters exchanged between CCD and the application is tabled below:

Name	Description
entity	ID of the calling entity. Valid entity identifiers are defined as C-macros in mconst.cdg.
direction	specifies whether the message goes UPLINK or DOWNLINK. This is necessary because the same PDU-Type can be used for both direction while the message structures can be different.
mBuf	specifies the bit stream buffer of the message. The elements of this structure are l_buf, o_buf and buf. The two first elements specify the length and offset of the bit stream buf. The o_buf component must be specified by the caller. The l_buf component is calculated by CCD in the older versions or is specified by the caller in the new version of CCD. The third element buf will contain the bit stream of the coded message.
mStruct	refers to the C-structure containing the C-representation of the message to be decoded. The type should be cast to UBYTE*. The first element must contain the message type (PDU) as a UBYTE value.
mId	specifies the PDU-Type of the bit stream. CCD reads this value as the message ID if it is not equal to 0xff or 0xfe. Normally this parameter is set to 0xff which means id is not defined. Thus CCD reads the pdu-type from the appropriate component of

*mStruct. The value 0xfe means „NO_TRACE" according to an internal convention of the CCD author.

Before entering the critical section CCD gets the semaphore protecting the shared buffers in case of multi threaded pre-emptive systems:

```
vsi_s_get (VSI_CALLER semCCD_Codec);
```

The usual trace functionality can be stopped for the Target systems utilized by ARM7. In this case if DEBUG_CCD and _TMS470 are switched on and if mId=0xfe, then the variable TraceIt will be set to 0. The result will be that TRACE_CCD() will return immediately after being called.

```
TraceIt = (mId EQ 0xfe);
```

After an error detection by CCD the function ccd_setError () is called. In some scenarios it may interrupt the normal operation of CCD through a long jump to the return line of ccd_encodeMsg() or ccd_decodeMsg(). The parameter needed for calling to setjmp() and longjmp() is **jmp_mark**. While encoding/decoding the same code line can be reached either through a normal operation or a long jump. The flag **jmp_mark_set** will be TRUE if the code line is reached by a long jump. At the beginning of ccd_codeMsg() it is set to FALSE because this code line is obviously reached through a normal behaviour.

```
jmp_mark_set = FALSE;
```

All global (shared) variables must be initialized:

```
bitpos = 0; pstructOffs  = 0; lastbytepos16 = lastbytepos32 = 0xffff;
bitoffs = mBuf->o_buf; bitbuf[bytepos] = 0; ccd_recurs_level = 0;
CCD_Error = ccdOK; aktEntity = entity; numCCDErrors[aktEntity] = 0;
```

The pointer `mBuf->buf` points to the buffer which will contain the bit encoded message. Before any manipulation of this buffer (also mstruct) CCD needs the following pointer assignments:

```
bitbuf = mBuf->buf;
pstruct = (mStruct EQ NULL) ? decMsgBuffer : mStruct;
```

The calling entity has already written the first message parts in the first bits of the buffer up to the bit numbered by `mBuf->o_buf`. Thus the bit position must be adjusted using the function bf_incBitpos(). It also notes the current byte position into the global variable bytepos.

```
bitoffs = mBuf->o_buf; bf_incBitpos (mBuf->o_buf);
```

Now the rest of the buffer needs to be set to 0 so that the old content has no chance to corrupt the new message. For this action we need to calculate the expected number of bytes in the bit coded message. This number can be calculated through the maximum bit size of the message, as in the older versions of CCD. Another way is to use directly the value `mBuf->l_buf` in case it is set by the entity. Not only in the mentioned calculation but also in the future operations it is important to know the message identifier. If the function parameter mId has a valid value (it is neither 0xff nor 0xfe) the temporary variable theMsgId will be set to this value. Otherwise CCD assumes that the next element to be read in the C-Structure is the valid message id and copies it.

```
if (mId NEQ 0xff AND mId NEQ 0xfe) theMsgId = mId;
else{ theMsgId = *pstruct; }
#if defined OLD_CCD
  maxBytes = (mcomp[mmtx[entity][theMsgId][direction]].bSize>>3)+1;
#else
  maxBytes =  mBuf->l_buf>>3;
#endif
...
#ifdef FAR_MODEL
  for (i = 0; i < maxBytes; i++)
    mBuf->buf[i+(mBuf->o_buf>>3)] = 0;
#else
  memset((UBYTE *) &mBuf->buf[(mBuf->o_buf>>3)], 0, maxBytes );
#endif
```

The first value CCD will write into the message buffer is the message identifier provided it is allowed to read this value from *pstruct. That means whenever the calling entity sets the parameter mId to 0xff, Bf_writeBits() will read the identifier from the address given by pstruct+_structOffs.

```
if (mId EQ 0xff)
  bf_writeBits (mi_length[entity]);
```

The coding can be processed further only if the key information, e.g. theMsgId, have valid values. Otherwise the coding is interrupt with an error report (ERR_INVALID_MID). The obtained Semaphore will then be released.

```
if (theMsgId > MAX_MESSAGE_ID OR mmtx[entity][theMsgId][direction] EQ NO_REF)
  ccd_setError (ERR_INVALID_MID, BREAK, (USHORT) theMsgId, (USHORT) -1);
#ifdef SHARED_CCD
  vsi_s_release (VSI_CALLER semCCD_Codec);
#endif
  return CCD_Error;
```

Three further initialization steps are needed before the rest of the message can be coded by ccd_encodeComposition(). The first step is somehow an initialization step for the UPN calculator implemented by calcUPN(). The second step is related to clearing the stack of the UPN calculator and to the initializing of the table iei_ctx[].

```
ST_CLEAR;
cdc_GSM_start ();
```

The third step concerns the usage of long jumps. If this code line is reached through a normal operation (jmp_ret=0) then the coding will be continued by a call to order to ccd_encodeComposition(). Otherwise

an error will be reported, the obtained semaphore will be released and the function ccd_codeMsg returns CCD_ERROR. In the latter case CCD calculates the length of the so far coded message in the buffer and writes it into the structure member mBuf->l_buf.

```
jmp_ret = setjmp (jmp_mark);
if (jmp_ret EQ 0){
  jmp_mark_set = TRUE;
  ccd_encodeComposition (mmtx[entity][theMsgId][direction]);
}
mBuf->l_buf = (USHORT) bitpos - (USHORT) mBuf->o_buf;
vsi_s_release (VSI_CALLER semCCD_Codec);
return CCD_Error;
```

7.2.5 ccd_decodeMsg

Because of the analogy between ccd_codeMsg() and ccd_decodeMsg() we leave this function undiscussed.

7.2.6 ccd_encodeComposition

ccd_encodeComposition() is an internal part of CCD which is called for encoding of IEs of a message. The only parameter of this function is: mcompRef. It refers to the appropriate entry for the IE in the mcomp table.

The function may call itself recursive for nested IEs. The global variable ccd_recurs_level helps to mark the current level in case of nested calls. For each message coding this variable is set to 0 by ccd_codeMsg() and can be increased or decreased only by ccd_encodeComposition().

```
ccd_recurs_level++;
```

The coding of the Subelements of the IEs in each recursion level will be processed in a while loop.

```
while (elemRef < lastElem) {...}; ccd_recurs_level--;
```

The limits are given by the (melem table) reference number of the first and the last element. Both need to be calculated using the members componentRef and numOfComponents of the mcomp table. For each message the first entry in mcomp (ccd_recurs_level = 0) belongs to the IE named message type. Since message type has been processed before calling ccd_encodeComposition() we use the next number for in this case; that is componentRef+1.

```
lastElem = mcomp[mcompRef].componentRef+ mcomp[mcompRef].numOfComponents;
elemRef  = mcomp[mcompRef].componentRef+ ((ccd_recurs_level EQ 1) ? 1 : 0);
```

Within the while-loop the IEs are divided in two categories. One is called to be an extended octet group, the other one not. For the latter the structure member extGroup is a white space character. For the first category we need to distinguish between the position of the subelement in the whole IEs, e.g. wether it is the first, the last or a middle octet. This issue is cleared by using the characters +, -, *, ! and #.

```
if (melem[elemRef].extGroup NEQ ' ') { switch (melem[elemRef].extGroup) {...}
} else {...}
```

For non-extended-group IEs then the corresponding coding function is called. After that the while-loop may be broken if there is no further element to be coded.

```
codecRet = codec[melem[elemRef].codingType][ENCODE_FUN](mcompRef, elemRef);
if (codecRet NEQ 0x7f) { elemRef += codecRet; }
```

For the extended octet groups the coding process has little differences depending on the member extGroup. Here we discuss each case of the above switch statement.

```
case '+':
```

To be continued...

File under construction at this point!

8 Technical information

The supported working environment for building CCD libraries is currently a reasonable command line shell, e.g. 4nt.exe, and gnumake. The CCD own source and header files are all collected in a directory called ccd. The name of files are given in the previous chapter.

The header files in the ccd directory are:
bitfun.h, ccd.h, ccdtable.h, ccddata.h and ccd_codingtypes.h

The rest of the required header files, besides those of the C standard library, are:

ccdapi.h, typedefs.h and vsi.h.

These files are shared between CCD and applications, including test applications. That is why they are located in a central directory called "../inc" relative to the directory of CCD own files.

At the compile time of CCD library there is no dependency to the group of CCDDATA (*.cdg or *.val) files.

Using gnumake rules the final part of the makefile script for building of CCD objects or library may look like this[3]:

```
CCD_OBJS := $(OBJDIR)/ccd.$(OBJTAIL) $(OBJDIR)/bitfun.$(OBJTAIL) \
              $(OBJDIR)/cdc_std.$(OBJTAIL) $(OBJDIR)/cdc_com.$(OBJTAIL)...

$(CCD_OBJS) : $(OBJDIR)/%.obj : $(CCD_SRC)/%.c
        $(CC) $(COPTSNF) $<

CCD_LIB  :=  $(LIBDIR)/ccd$(XXX).$(LIBEXT)

ccd.lib : $(CCD_LIB)

$(CCD_LIB) : $(COPTS_FILE) $(CCD_OBJS) makefile
        $(LIB)  $(CCD_OBJS)
```

8.1.1 Makefile variables

In order to specify the name of the different variations of ccd.lib suffixes are appended to that name. For example ccd_na7_tr_db.lib means the debuggable version with trace outputs and for the RTOS nucleus on ARM7 target. The different elements of the name suffix are:

trTail = _tr	for TRACE=1
dbTail = _db	for DEBUG=1
psTail = _ps	for MEMSUPER=1
tTail = _na7	for TARGET=nuc and PLATFORM=arm7
tTail = _na9	for TARGET=nuc and PLATFORM=arm9

[3] This is actually part of the new version of the makefile for ccd under gpf/ccd/.

tTail = _npc for TARGET=nuc and PLATFORM=pc

And the final suffix is XXX.

```
XXX:=$(tTail)$(trTail)$(dbTail)$(psTail)
```

The full name of the built library will then be

```
CCD_LIB=$(LIBDIR)/ccd$(XXX).$(LIBEXT)
```

Where the extension LIBEXT is either lib or dll.

Different projects for different RTOSes use different compilers and linkers. With the RTOS Nucleus the compiler is cl470 and the archiver is ar470.exe. For the work on the Nucleus emulation on win32 cl.exe is both the compiler and the linker.

The preprocessor macros are the followings:

_TMS470	specifies build for target board
NEW_FRAME	for integrating to the new generation of frame
M_INTEL	for using intel processor on the target board
SHARED_VSI	for using vsi calls in a preemptive multithreaded system
SHARED_CCD	for using CCD in a preemptive multithreaded system

The compiler and linker (or archiver) options for each compiler are listed below:

Nucleus	
Compiler (cl470) Options	**Comment**
`-me`	Produce code for little-endian configuration.
`-mt`	Generate 16-bit code.
`-pw2`	Enable all warning messages.
`-q`	Suppress banners and progress information from all the tools.
`-x`	Expanded inline functions.
`-mw`	
`-o`	

`-g`
`-mn`
`{only if the environment variable DEBUG=1}`
	Generate symbolic debugging directives.
	Reenable the optimizations disabled by the –g option.

`-fr $(OBJDIR)`	Put the object files into this directory.
`-fo $@`	Interpret the parameter as the name of the output file.
`-c`	Suppress the linking option.
`-I".\" -I"..\INC"`	Include header files related to CCD.
Linker (ar470) Option	**Comment**
`-rq`	

Nucleus emulation on win32	
Compiler option	**comment**
`/c`	Just compile, don't link. Leave output in a .obj file
`/nologo`	Disables the display of the copyright banner.
`/W3`	Set the most sensitive warning level recommended for production purposes.
`/GX`	enable synchronous exception handling with the assumption that extern C functions never throw an exception.
`/Zp1`	Packs structures on 1-byte boundaries
`/MD`	Use multithread- and DLL-specific versions of the run-time routines
`/FR"..\temp\nucwin"`	create an .SBR file containing symbolic information.
`/Fd"..\temp\nucwin"`	Use the given name for the program database (PDB).
`/Fo./$(OBJDIR)/`	Put the object files to this directory.

/I "c:\programme\devstudio\vc\include"	Include the header files of the C standard library
/I /I "..\INC"	Include the header files related to CCD
/D "_DEBUG" /DEBUG /Z7 {only if the environment variable DEBUG=1}	Produce debug information in .obj file.
Linker Option	Comment
/nologo	Disables the display of the copyright banner.
/subsystem:console	Produce a console application.
/machine:I386	Set the target platform for the program.
/out:$@	Set the path to the directory in which the stub, header, and switch files are generated.
/debug {only if the environment variable DEBUG=1}	creates debugging information for the .EXE file or DLL
/pdb:none	put old-style debugging information into the .EXE file or DLL.

Win32	
Compiler option	**comment**
/c /nologo /W3 /GX /Zp1 /MD /FR"..\temp\win32" /Fd"..\temp\win32" /Fo./$(OBJDIR)/ /I ... /D "_DEBUG" /DEBUG /Z7	See descriptions above !
Linker Option	Comment
/nologo	Disables the display of the copyright banner.
/subsystem:console	Produce a console application.
/machine:I386	Set the target platform for the program.
/out:$@	Set the path to the directory in which the stub, header, and switch files are generated.
/force:unresolved	create an output file whether or not LINK finds an undefined symbol.
/debug {only if the environment variable DEBUG=1}	creates debugging information for the .EXE file or DLL
/pdb:none	put old-style debugging information into the .EXE file or DLL.

Appendices

A. Acronyms

DS-WCDMA Direct Sequence/Spread Wideband Code Division Multiple Access

B. 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/>