Chapter

59

ASN.1 Encoding and Decoding in the SDL Suite

This chapter is a reference manual for the ASN.1 encoding and decoding support in the SDL Suite. The support will help you to create applications and simulators that can both produce and decode protocol data units with bit patterns according to ASN.1 specifications and ASN.1 encoding rules. The SDL Suite supports BER (Basic Encoding Rules) and PER (Packed Encoding Rules).

In order for you to fully take advantage of this reference chapter, you should be familiar with the SDL Suite and the basics of ASN.1.

> **Note:**
>
> **This chapter is not intended to describe the ASN.1 syntax or the general theory about ASN.1 encoding and decoding. This information can be found in the standard documents from ITU and from books about ASN.1.**

Introduction

Supported Standards

The following standards are supported:

- ASN.1 according to ITU X.680, X.681, X.682 and X.683, 1997 version
- BER (Basic Encoding Rules) according to ITU X.690, 1997 version
- Basic PER (Packed Encoding Rules) according to ITU X.691, 1997 version, both aligned and unaligned variants.

Restrictions are listed in the Known Limitations chapter in the Release Guide.

Overview

The Abstract Syntax Notation One (ASN.1) is a notation language that is used for describing structured information that is intended to be transferred across some type of interface or communication medium. It is especially used for defining communication protocols.

SDL is suitable for specifying the semantic actions of a protocol, something that ASN.1 does not cover.

By using ASN.1 data types in the implementation of your application, you will optimize your development process. The following list displays some of the advantages of ASN.1:

- ASN.1 is a standardized, vendor-, platform- and language independent notation.
- A vast number of telecommunication protocols and services are defined using ASN.1. This means that pre-defined ASN.1 packages and modules are available and can be obtained from standardization organizations, RFCs, etc.
- When ASN.1 data types are transmitted over computer networks, their values must be represented in bit-patterns. Encoding and decoding rules determining the bit-patterns are already defined for ASN.1. The SDL Suite supports BER and PER encoding.

The SDL Suite support for ASN.1 consists of two major parts:

- An ASN.1 to SDL translator, ASN.1 Utilities, which allows you to use the ASN.1 types in your SDL systems.
- ASN.1 coders that will automatically encode your SDL values to bit-patterns and decode from bit-patterns to SDL values.

You can access the ASN.1 coders from SDL diagrams as well as from the C code.

Before you start the encoding and decoding steps, you must perform some preliminary steps. These are presented in chapter 8, *Tutorial: Using ASN.1 Data Types, in the SDL Suite Getting Started*.

Related Documents

Additional information about using ASN.1 and about encoding and decoding can be found in:

- chapter 8, *Tutorial: Using ASN.1 Data Types, in the SDL Suite Getting Started*
- chapter 14, *The ASN.1 Utilities*
- chapter 58, *Building an Application*
- chapter 66, *The Cmicro SDL to C Compiler*

Basic Concept

ASN.1 is suitable for specifying the information in a protocol and SDL is suitable for describing the semantic actions of a protocol. They are often used by standard bodies to specify protocol standards.

One of the main ideas behind ASN.1 is that you should be able to work with information in the protocol in the same way as you work with types and values in your implementation language. It should not be more difficult to create a bit-pattern in a protocol data unit then it is to assign a value to a variable. The encoder and decoder functions will automatically deal with all the details about bit-patterns and bit-layouts.

ASN.1 Utilities

The ASN.1 to SDL translator, *ASN.1 Utilities*, translates ASN.1 types and constants into SDL types and constants. An ASN.1 module generates an SDL package. This allows you to work with translated types and constants in the same way as any SDL type or constant. For instance, you can use them when you declare variables and when you declare signal parameters.

Information about ASN.1 to SDL translation and C representation of SDL data types is available in:

- chapter 14, *The ASN.1 Utilities*
- chapter 2, *Data Types, in the SDL Suite Methodology Guidelines*

Encoding and Decoding

Encoding and decoding functionality consists of several parts.

First of all, it is the encoding and decoding algorithms themselves. You encode a variable or signal parameter into a bit-pattern by calling the encoder functions. You decode a bit-pattern into a variable or signal parameter by calling the decoder functions.

Another integral part of the coding functionality is the buffer handling. The decoder function takes the bit-pattern from the buffer and creates values for the application. The encoding function takes the internal value and encodes it into a sequence of bits in the buffer.

Error handling functionality is also present in the coding framework. Several types of errors can occur when encoding and decoding, such as buffer errors, incorrect values, decoding errors. Error handling functions allow to handle all possible erroneous situations.

Coding Access Interfaces

The coding functionality has got several access interfaces that allow you to access it in different ways.

It is possible to access encoding and decoding functions, call buffer functions and check error codes directly from the SDL diagram or from the C code in the environment files. For the C access interface, calls to encoder and decoder functions together with the appropriate buffer access procedures can be generated automatically if a few options in the user interface has been selected.

Basic SDL Interface and Extended SDL Interface

In SDL, you can use the basic interface or the extended interface. The basic interface is easy to use but gives you no possibility to control the buffer management. The extended interface gives you flexibility that the basic interface lacks.

The basic interface contains encode and decode functions which operates at SDL octet strings. The extended interface adds the possibility to access the buffer interface from within SDL and operates at a new type call CoderBuf. Furthermore, the extended interface contains a new type called CustomCoderBuf with which you could implement your own mapping between SDL and the encode and decode functions.

Within an SDL system, it is only possible to use one type of SDL interface. Either the SDL Basic Interface, the SDL Extended Interface with CoderBuf or the SDL Extended Interface with CustomCoderBuf.

C Code Interface

The coder library contains encoding and decoding functions and also several help functions for buffer management, error management and printing information. If you choose to call encoding and decoding from outside the environment files, note that:

- You need a file named `<ASN.1 module>.ifc` with necessary C declarations of types, literals and operators. The SDL to C Compiler generates this file if you have added the ASN.1 module to your SDL system with a use package clause.

- You need generated C files and static kernel C files. These files will be present if you have generated an SDL system with your ASN.1 module.

General information about using .ifc files and writing C code with SDL is available in:

- chapter 58, *Building an Application*
- chapter 66, *The Cmicro SDL to C Compiler*

Solution

This section describes the structure and design of the encoding and decoding support in the SDL Suite. It is intended to give you an understanding of the solution.

Functionality

The main goal is to help you to develop executable files, applications or simulators, that can:

- produce BER/PER bit patterns
- consume and decode BER/PER bit patterns
- be configured to work with user specific environment (memory management, buffer management and error handling)

ASN.1 coder framework contains the following functional modules:

- Encoding and decoding functionality (see “Encoding and Decoding Functionality” on page 2764)
- Buffer Management System (BMS) (see “Buffer Management System” on page 2776)
- Memory Management System (MMS) (see “Memory Management System” on page 2792)
- Error Management System (EMS) (see “Error Management System” on page 2796)
- Debug print opportunities (see “Printing Opportunities” on page 2811)

All these modules will be described in more details in this chapter.

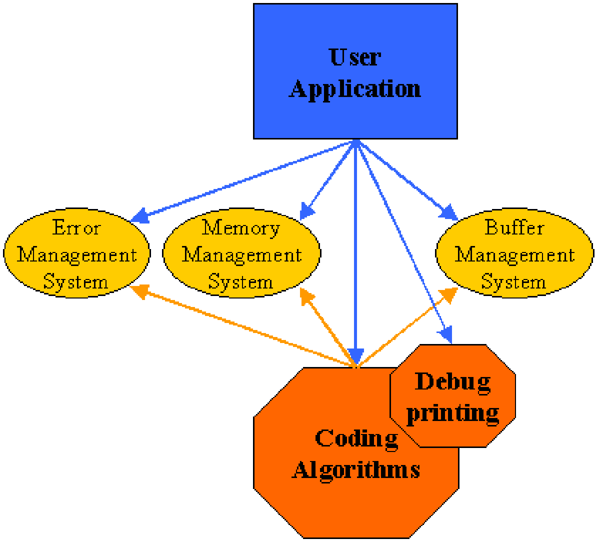

Figure 495: Relationship between functional modules of the encoding/decoding solution

Functionality Access Interfaces

When using the coder functionality, you can choose between different access interfaces. They all share the same implementation.

- **Basic SDL interface**

  SDL operators that put bytes in octet strings and read bytes from octet strings. Buffer access functions are not visible in this SDL interface. For more information see "Encoding/decoding the SDL Basic Interface" on page 2768.

- **Extended SDL interface**

  SDL `encode` and `decode` operators that put bytes in the types CoderBuf and CustomCoderBuf. Parts of the buffer management is visible in the interface. For more information see "Encoding/decoding the SDL Extended Interface" on page 2770.

- **C code interface**

  This interface is used when calling the coders from C code, for example in the environment files. For more information see "C Encoding and Decoding Interface" on page 2764.

The SDL and C interfaces are not identical. They are adapted to the properties of the languages.

It is preferred to access the coders from SDL when the encoded bit-pattern must be further processed in the SDL system. One interesting example is when the bit pattern is sent in a signal to another process.

It is preferred to access the coders in C-code in the environment files when the SDL-system does not process the bit pattern internally, for example when sending it to a protocol in the environment.

Encoding and Decoding Functionality

Encoding and Decoding Functions

An encoding function encodes an SDL variable or signal parameter into a bit-pattern according to BER or PER. A decoding function decodes from a PER or BER bit-pattern to an SDL variable or signal parameter. You can access the encoding and decoding functions either from the C code (see "C Encoding and Decoding Interface" on page 2764) or from SDL diagrams (see "SDL Encoding and Decoding Interfaces" on page 2768).

C Encoding and Decoding Interface

The most common example of implementing coders in C is in the environment file. The SDL to C Compiler automatically generates an environment file with all needed calls. The coders can be accessed from any function or module implemented in C code.

There is one set of functions for BER and one set of functions for PER. The functions are accessed by using macros `BER_ENCODE` and `BER_DECODE`, respectively `PER_ENCODE` and `PER_DECODE`.

There is also a set of common functions which choose encoding rules dynamically according to the parameter written to the buffer by function `BufSetRule(buffer, rule)`, see "Buffer Management Functions" on page 2779. The functions are accessed by using macros `ASN1_ENCODE` and `ASN1_DECODE`.

Example 453 Dynamic encoding rules configuration ––––––––––––––

```
   BufInitWriteMode(buf);
   BufSetRule(buf, er_PER | er_Aligned);
```

/* PER Aligned will be applied */

```
   ASN1_ENCODE( buf, (tASN1TypeInfo *)&yASN1_MyType1,
                (void *)&((yPDP_Sig1)(*S))->Param1));
   BufSetRule(buf, er_BER | er_Definite);
```

/* BER Definite will be applied */

```
   ASN1_ENCODE( buf, (tASN1TypeInfo *)&yASN1_MyType2,
                (void *)&((yPDP_Sig2)(*S))->Param1));
   BufCloseWriteMode(buf);
```

–––

All the encode and decode macros, ASN1_ENCODE,ASN1_DECODE, BER_ENCODE, BER_DECODE, PER_ENCODE and PER_DECODE have three parameters:

- **First parameter**:

  The first parameter is a reference to a buffer of type tBuffer (see "Types and definitions" on page 2777). During encoding the resulting bit pattern will be stored in a memory section associated with the buffer. The buffer must be in the write mode when the encode function is called, because it writes bytes to the buffer. During decoding a bit pattern will be read from a memory section associated with the buffer. The buffer must be in read mode when the decode function is called, because it reads bytes from the buffer. For more information about buffer handling see "Buffer Management System" on page 2776.

- **Second parameter**:

  Pointer to ASN.1 type information (see "ASN.1 Type Information Generated by ASN.1 Utilities" on page 2767). All information that is needed about the type in the ASN.1 specification is represented in this type information. The reference to type information is of type tASN1TypeInfo*. The type information structure is a global variable with a name yASN1_<type name>. The reference to the type information structure is then (tASN1TypeInfo*)&yASN1_<type name>.

- **Third parameter**:

  Memory pointer to variable or signal parameter to encode or decode. The memory pointer is of type void* .

The buffer macro returns an error code which is a literal from an enumerated type (see "Error codes" on page 2799). The return value indicates whether the encoding or decoding procedure was successful or not. If this value is ec_SUCCESS, then procedure was successful.

Example 454: BER encoding of signal parameter ––––––––––––––––

In ASN1:

```
Message ::= SEQUENCE {
                       id    INTEGER,
                       info  INFOTYPE
                     }
```

In SDL:

```
SIGNAL Sig2(Message);
```

In xOutEnv:

```
BufInitWriteMode(buf);
BER_ENCODE( buf,
           (tASN1TypeInfo *)&yASN1_Message,
            (void *)&((yPDP_Sig2)(*S))->Param1));
BufCloseWriteMode(buf);
```

Example 455: PER encoding of signal parameters

In xOutEnv:

```
BufInitWriteMode(buf);
PER_ENCODE( buf,
            (tASN1TypeInfo *)&yASN1_Message,
            (void *)&((yPDP_Sig2)(*S))->Param1));
BufCloseWriteMode(buf);
```

Example 456: BER decoding of signal parameters

In xInEnv:

```
BufInitReadMode(buf);
BER_DECODE( buf,
           (tASN1TypeInfo *)&yASN1_Message,
            (void *)&((yPDP_Sig2)(S))->Param1));

BufCloseReadMode(buf);
```

Example 457: PER decoding of signal parameters

In xInEnv:

```
BufInitReadMode(buf);
PER_DECODE( buf,
           (tASN1TypeInfo *)&yASN1_Message,
            (void *)&((yPDP_Sig2)(S))->Param1));

BufCloseReadMode(buf);
```

ASN.1 Type Information Generated by ASN.1 Utilities

The encoder and decoder functions need information about the types from the ASN.1 specifications. This information is stored in type information structures, called type information nodes or simply type nodes. Almost all information about the types that you can find in the ASN.1 specifications is stored in the type information nodes.

The internal structure of the type information nodes is important in an implementation of encoding and decoding functions, but not from the application that calls the functions. There, it is only important to find a reference to a node. The internal details of the type nodes are not described in this section.

The type information nodes are represented in C code as a set of global read-only variables. The information in a node cannot be changed during execution. The nodes are linked to each other by using memory references. The type nodes can be declared as `const` variables by defining a macro, which means that in some application they can be moved to program memory or read only memory. For more information, see “Compilation switches” on page 2817.

The name of a node is `yASN1_<type name>`. A reference to a type node is simply a memory pointer to the type node cast to a basic type node declaration.

Example 458: Reference to a type node –––––––––––––––––––––––

If the type name is `Pdu1` in the ASN.1 specification, the reference to the type node is:

```
(tASN1TypeInfo*)&yASN1_Pdu1.
```

–––

The nodes reflect the information in ASN.1 specifications and must be generated by ASN.1 Utilities. ASN.1 Utilities parses and analyzes the ASN.1 information and generates C code containing type nodes. One .c-file and one .h file per ASN.1 module is generated and the files are named `<module name>_asn1coder.c` and `<module name>_asn1coder.h`.

SDL Type and Operator Information Generated by the SDL to C Compiler

The encoder and decoder functions also need information generated by the SDL to C compiler, for example declarations of operators and literals for manipulating the translated ASN.1 types. This information is also stored in the type nodes.

ASN.1 utilities generates one SDL package for each ASN.1 module. The SDL to C Compiler generates one `.c` and `.h` file for each package, with type and operator information, together with one `.ifc` file for an external view of the types and operators. For more detailed information about the C code generation see "System Interface Header File" on page 2704 in chapter 58, *Building an Application*.

The ASN.1 type information files includes the `<package>.ifc` files. All relationships are resolved during compile and link time and the information is read-only at execution time.

SDL Encoding and Decoding Interfaces

In the SDL interface, the ASN.1 type information pointer is not seen. It is implicit and derived from the type of the memory pointer.

There are two types of coder interfaces available from within an SDL system: SDL basic interface (see "Encoding/decoding the SDL Basic Interface" on page 2768) and SDL extended interface (see "Encoding/decoding the SDL Extended Interface" on page 2770). In the SDL basic interface, the buffer reference is replaced with an octet_ string and the encoded bit-pattern is put directly in a variable of octet_string type. In the SDL extended interface, the buffer reference is CoderBuf or CustomCoderBuf.

Encoding/decoding the SDL Basic Interface

You can encode and decode directly from the SDL diagram by using the operators `encode` and `decode`.

The syntax of the `encode` function is:

```
result:= encode(encoded_octet_string, ASN1_variable)
```

The two parameters of the `encode` function are defined as:

- `encoded_octet_string` is an `Octet_string` that contains the encoded value of the ASN1_variable if the encoding was successful.

- `ASN1_variable` is the ASN.1 variable to be encoded.

The function returns an integer, `result`. If the value is equal to `ec_SUCCESS` the encoding was successful. For more information about return status see “SDL error interface” on page 2799.

The encoded value in the octet string could then be used in a signal sent to an other part of the system or to the environment.

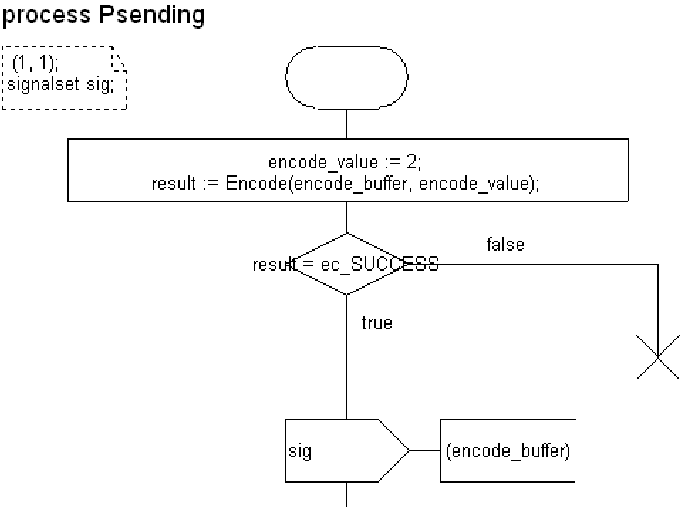

Figure 496 Example of a encode call in an SDL diagram

The syntax of the `decode` function is:

```
result:= decode(octet_string_to_decode,
ASN1_variable)
```

The two parameters of the `decode` function are defined as:

- `octet_string_to_decode` is an `Octet_string` that contains the encoded value to decode.
- `ASN1_variable` is the ASN.1 variable that will contain the result after the decoding was performed.

The function returns an integer, `result`. If the value is equal to `ec_SUCCESS` the encoding was successful. For more information about return status see "SDL error interface" on page 2799.

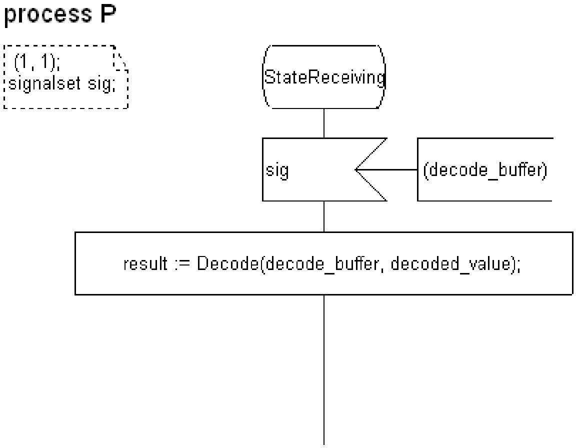

Figure 497 Example of a encode call (SDL Basic Interface)

Encoding/decoding the SDL Extended Interface

As with the Basic SDL Interface you can encode and decode directly from the SDL diagram by using the operators `encode` and `decode`.

The syntax of the `encode` function is:

```
result:= encode(CoderBuf, ASN1_variable)
```

The two parameters of the `encode` function are defined as:

- `CoderBuf` is a type defined in the file `CoderBuf.sdl` . It is a mapping of the buffer management C-interface (see "C interface to buffer management system" on page 2777). For more information about operations at CoderBuf see "SDL CoderBuf interface" on page 2788.
- `ASN1_variable` is the ASN.1 variable to be encoded.

The function returns an integer, `result`. If the value is equal to `ec_SUCCESS` the encoding was successful. For more information about return status see "SDL error interface" on page 2799.

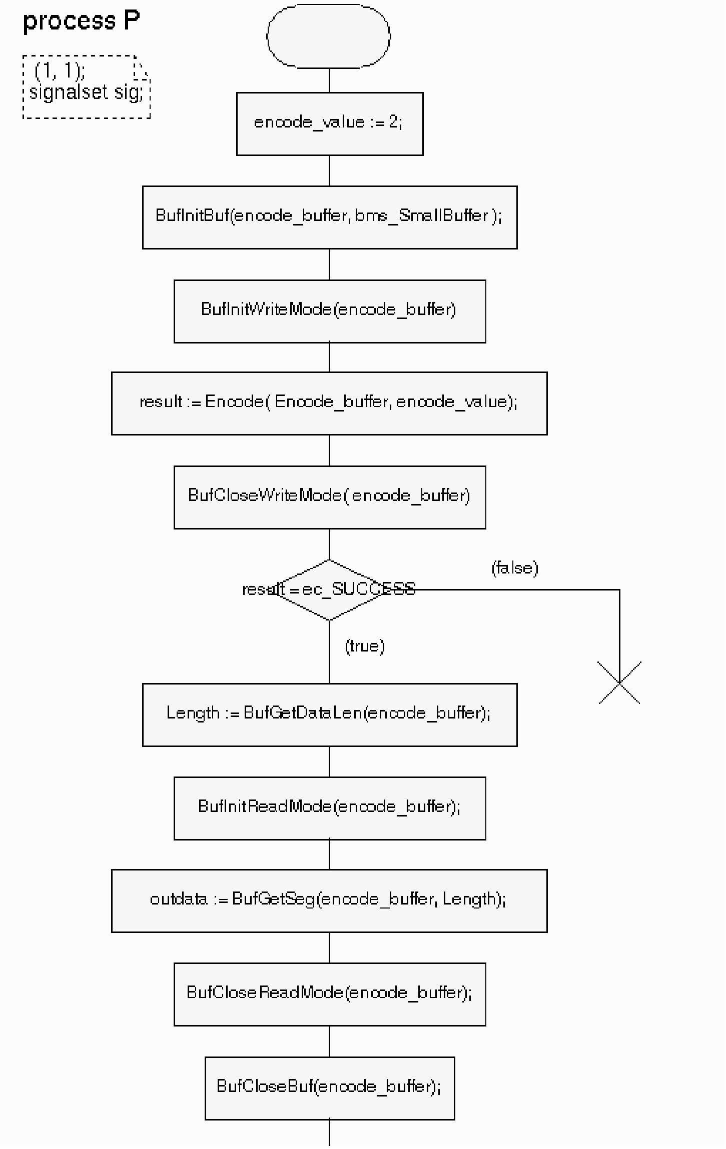
process P
(1, 1);
signalset sig;
encode_value := 2;
BufInitBuf(encode_buffer, bms_SmallBuffer);
BufInitWriteMode(encode_buffer)
result := Encode(Encode_buffer, encode_value);
BufCloseWriteMode(encode_buffer)
result = ec_SUCCESS
(false)
(true)
Length := BufGetDataLen(encode_buffer);
BufInitReadMode(encode_buffer);
outdata := BufGetSeg(encode_buffer, Length);
BufCloseReadMode(encode_buffer);
BufCloseBuf(encode_buffer);

Figure 498: Example of en encode call (SDL Extended interface)

The syntax of the `decode` function is:

```
result:= decode(CoderBuf, ASN1_variable)
```

The two parameters of the `decode` function are defined as:

- `CoderBuf` is a type defined in the file `CoderBuf.sdl`. It is a mapping of the buffer management C-interface (see "C interface to buffer management system" on page 2777). For more information about operations at CoderBuf see "SDL CoderBuf interface" on page 2788.
- `ASN1_variable` is the ASN.1 variable that will contain the result after the decoding was performed.

The function returns an integer, `result`. If the value is equal to `ec_SUCCESS` the encoding was successful. For more information about return status see "SDL error interface" on page 2799.

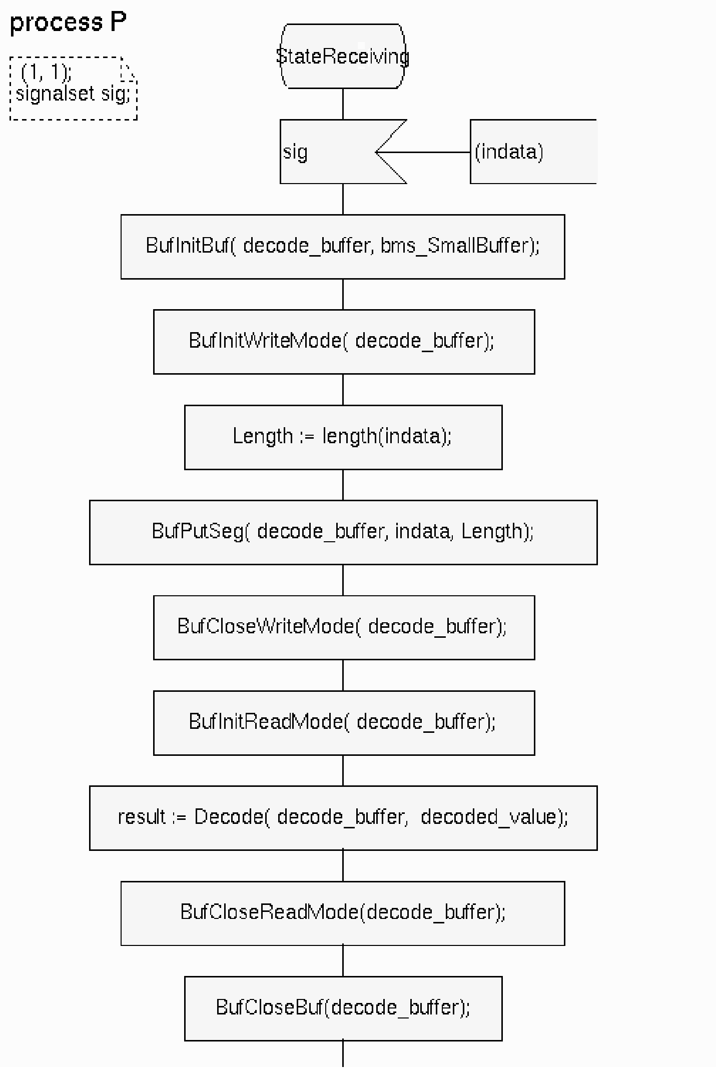
process P
(1, 1);
signalset sig;
StateReceiving
sig
(indata)
BufInitBuf(decode_buffer, bms_SmallBuffer);
BufInitWriteMode(decode_buffer);
Length := length(indata);
BufPutSeg(decode_buffer, indata, Length);
BufCloseWriteMode(decode_buffer);
BufInitReadMode(decode_buffer);
result := Decode(decode_buffer, decoded_value);
BufCloseReadMode(decode_buffer);
BufCloseBuf(decode_buffer);

Figure 499: Example of decode call (SDL Extended Interface)

How to Include the Extended SDL Interface in Your System

Add the package in the Organizer. In the directory `<installation_directory>/sdt/include/ADT/` there are two versions of the BufferInterface package. One for C, `CoderBuf.sun`, and one for C++, `CoderBufCPP.sun` . When the package is included in the Organizer, add a use statement of the BufferInterface in your system.

> **Note: Types in the BufferInterface package**
>
> In the BufferInterface package BasicCTypes and CPointer or the corresponding C++ versions of those files are included. This must be considered when using `cpp2sdl` together with your system so that those files are included only once.

The Extended SDL Interface in the Simulator

You can display the value of a CoderBuf and change it when running the Simulator by using the commands examine-variable and assign-variable. If you have written your own buffer you can add information in the provided files to be able to use them in the Simulator, see below.

Examine CoderBuf

Command: ex-va encode_buffer

encode_buffer (CoderBuf) = bms_UserBuffer(NM)3’020102’H

bms_UserBuffer is the buffer type, (NM) shows the current mode of the coderbuffer(NoMode, ReadMode or WriteMode), 3 is the length of the coder buffer value in bytes and ’020102’H is the value of the coder buffer in octet string form

Limitations:

Examining the value when the coder buffer is in Write Mode is not possible.

If the user has defined a new buffer type other than Small Buffer and User buffer the buffer type will not be displayed.

Example:

encode_buffer (CoderBuf) = BufferType(NM)3’020102’H

To display the right buffer type the user has to do some modifications in the write function (yWri_CoderBuf) in the file CoderBuf.sdl.

Only the first 100 bytes of a coder buffer value are displayed during simulation.

Assign new values to a coder buffer:

Command: ass-va encode_buffer 2 '0101'H ReadMode(RM)

Value assigned:

encode_buffer (CoderBuf) = bms_UserBuffer(RM)2'0101'H

When you assign value to a coder buffer:

Command: ass-va <length> <value> <Mode>

Length is the length of the value in bytes. Value is the new value (octet string). Mode, the user sets the mode of the coder buffer (Read, Write and No).

In order for the assignment to be successful the user has to give the two first parameters; otherwise the command will not be executed. An error message will be displayed to the user.

If the length of the value passed as parameter is less than the length passed as parameter, extra zeros (0) are added to the octet string to match the length passed by the user. If the length of the value passed as parameter is greater than the length passed as parameter, then the octet string to be assigned will be cut to match the length.

If the user does not choose the buffer mode, the buffer mode will be set to No Mode (default).

Limitations:

If the user has defined a new buffer type other than Small Buffer and User buffer, the user has to do some modifications in the read function (yRead_CoderBuf) in CoderBuf.sdl in order to initiate the new coder buffer to be assigned to the old coder buffer with right coder buffer type.

The maximum size of the coder buffer is assumed to be 128 Mega bytes.

Buffer Management System

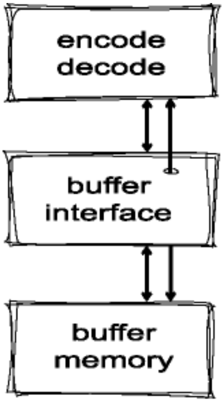

Figure 500: Relationship between coders and buffer management

The buffer interface is a set of functions that operates with data. Buffer management stores data in the buffer memory. Data is presented as `byte`, `bit` or `character segment`. Available basic operations are `get`, `put`, `peek` and `skip`. The buffer interface also opens a possibility for direct access to buffer memory from BMS environment (from encode and decode). According to this there is direct access to the buffer memory which presented as one continuos piece.

The buffer management functions have an open design, where it is possible to choose between different buffer implementations with different characteristics. This choice is available in runtime mode and also within compilation by switches defined in "Buffers configuration" on page 2822. This is possible due to an open and specified buffer interface that all encode and decode functions use and that your SDL specifications and C functions should use as well. The interface is specified as a set of C macros. See "C interface to buffer management system" on page 2777 for more information. The buffer management implements all the macros in the interface.

The buffer management is accessed from an SDL system by using operators in the buffer management Abstract Data Type. The operators are mapped to the open buffer management interface.

The only buffer management implementation at present in the SDL Suite is the small buffer (see "Small Buffer Implementation" on page 2788).

C interface to buffer management system

This chapter describes the open buffer interface. This is the C access interface to the buffer management.

Types and definitions

The buffer access type is `tBuffer`. `tBuffer` is defined as a pointer to a structure `tCoder`.

Example 459: Buffer as a reference ––––––––––––––––––––––––––––––

```
tBuffer buf = NULL;
BufInitBuf(buf, bms_SmallBuffer);
```

`tCoder` is used as access type only in tiny mode (see Example 469 on page 2822).

BMS introduces several specific types which are described below.

tBMSLength

BMS length type is defined as `unsigned long`.

tBMSUserMemory

This type is structure describing information about memory provided by the user. It contains three fields:

- `MemSize` of type `tBMSLength` - size of the provided memory
- `DataLength` of type `tBMSLength` - size of the data written to the provided user memory
- `MemPtr` of type `void *` - pointer to the provided memory

This type is used as parameter type in the buffer interface functions `BaseBufInitBufWithMemory` and `BaseBufCloseBufToMemory`.

tBMSBufType

This type is an enumeration describing all following possible types of buffers. Two types of buffers are available:

- `bms_SmallBuffer` - predefined buffer type (see "Small Buffer Implementation" on page 2788)
- `bms_UserBuffer` - type denoting user defined buffer (see "User defined Buffer Handling" on page 2790)

This type is used for specifying the type of buffer which will be used.

tERRule

This type is an enumeration describing possible kinds of encoding rules. It contains basic rules (BER, PER, etc.) as well as their different variations:

- `er_BER`
- `er_Definite`
- `er_Indefinite`
- `er_PER`
- `er_Aligned`
- `er_Unaligned`
- `er_NoEndPad` - PER without end padding
- `er_UER` - User defined encoding rules
- `er_NoRule`

The enumeration values are regarded as bit masks for different kinds of encoding rules, for example, PER Aligned can be specified with `er_PER | er_Aligned`; expression `( SpecifiedRule & ( er_PER | er_Aligned ) )` will check if the specified type of encoding rules `SpecifiedRule` is PER Aligned.

tBMSMode

This type is an enumeration describing possible modes for the buffer:

- `bms_ReadMode` - reading mode, writing is allowed for appending. See `BufSetAppendBufFunc( buffer, func )` for more information.
- `bms_WriteMode` - writing mode, reading is not allowed
- `bms_NoMode`

Buffer Management Functions

The buffer management functions put bytes and bits into a buffer which you can use to send over a protocol. The buffer management also provide several interfaces to operate with buffer memory and buffer handling.

The internal implementation of the buffer is not shown in the buffer interface, thus all manipulation of the buffer should be done using the macros. A buffer must be created before use. It must be opened before a read/write session and closed after a session. A buffer can be in either a read mode or a write mode, but not in both modes at the same time.

The open buffer interface macros are described below (parameter `buffer` in all functions is of type `tBuffer`, see "Types and definitions" on page 2777):

- `BufInitBuf( buffer, buffer_type )`

  This function is used to initialize the buffer management. Memory used by the buffer is allocated by `CUCF_ALLOC` function, so memory management used in the buffer handling can be configured by means of MMS configuration opportunities (see "User defined Memory Handling" on page 2794). The second parameter is the type of the buffer (see "tBMSBufType" on page 2777). The function returns error status (see "Error codes" on page 2799).

- `BufCloseBuf( buffer )`

  Closes a buffer and if the buffer has not been initialized by the user memory, releases buffer memory by using `CUCF_FREE`. The function returns error status (see "Error codes" on page 2799).

- `BufInitBufWithMemory( buffer, buffer_type, user_memory)`

  This function initializes the buffer with the user memory provided in the third parameter (see "tBMSUserMemory" on page 2777) without allocating any memory. If user memory contains any encoded data, the buffer can be used further by the decode function. The second parameter is the type of the buffer (see "tBMSBufType" on page 2777). The function returns error status (see "Error codes" on page 2799).

- `BufCloseBufToMemory( buffer, user_memory )`

This function closes the buffer. It does not release any memory, but returns it in the `user_memory` parameter of type "tBMSUserMemory" on page 2777 passed by reference. The function returns error status (see "Error codes" on page 2799).

- `BufGetMemory( buffer, length )`

  Returns pointer to the buffer memory. The second parameter is used for returning. It is pointer to the length of memory.

- `BufAppendMemory( buffer, append_length )`

  Used for the appending of the buffer memory with required `append_length`. Returns pointer to the beginning of the memory (root).

- `BufGetDataCurr( buffer, bit_position )`

  Returns current data pointer in the buffer handling. The second parameter is used for returning. It is a pointer to a char with current bit position in the buffer handling. This function is used by the decoder.

- `BufSetDataCurr( buffer, length, bit_position )`

  Used to increment current data pointer by `length` in the buffer handling and also sets current bit position to `bit_position` there. This function is used by the decoder.

- `BufGetDataEnd ( buffer, bit_position )`

  Returns end data pointer in the buffer handling. The second parameter is used for returning. It is a pointer to a char with the end bit position in the buffer handling. This function is used by the encoder and decoder.

- `BufSetDataEnd ( buffer, length, bit_position )`

  Uses to increment end data pointer by `length` in the buffer handling and also sets end bit position to `bit_position` there. This function is used by encoder.

- `BufCopyBuf( buffer_dst, buffer_src )`

  This function copies the source buffer `buffer_src` to target buffer `buffer_dst`. The function returns error status (see "Error codes" on page 2799).

- `BufInitReadMode( buffer )`

Prepares the buffer for read operations (read mode, see "tBMSMode" on page 2778) and puts the current pointer to the start of the buffer data. The buffer content is not changed or released. The function returns error status (see "Error codes" on page 2799).

- `BufCloseReadMode( buffer )`

  Closes read mode for the buffer. The buffer content is not changed or released. The function returns error status (see "Error codes" on page 2799).

- `BufCloseDeleteReadMode( buffer )`

  Closes read mode for the buffer and logically (without memory releasing) removes read bytes from buffer. Bytes that are not read are still in the buffer. The function returns error status (see "Error codes" on page 2799).

- `BufInitWriteMode( buffer )`

  Prepares the buffer for write operations (write mode, see "tBMSMode" on page 2778) and sets the current pointer to the start of the buffer. Previous information in buffer is logically lost (the bytes are still there but they are considered to be empty). The function returns error status (see "Error codes" on page 2799).

- `BufCloseWriteMode( buffer )`

  Closes write mode for the buffer. The buffer content is not changed or released. The function returns error status (see "Error codes" on page 2799).

- `BufGetBufType( buffer )`

  Returns the type of the buffer, see "tBMSBufType" on page 2777. It can be useful when the system supports more than one buffer interface and the type of the buffer is not obvious and can change dynamically.

- `BufGetDataLen( buffer )`

  Returns the byte length (see "tBMSLength" on page 2777) of all data that is physically present in the buffer.

- `BufGetDataBitLen( buffer )`

  Returns the bit length (see "tBMSLength" on page 2777) of all data that is physically present in the buffer.

- `BufGetReadDataLen( buffer )`

  Returns the byte length (see "tBMSLength" on page 2777) of the information which has been read from the buffer.

- `BufGetReadDataBitLen( buffer )`

  Returns bit length (see "tBMSLength" on page 2777) of the information which has been read from the buffer.

- `BufGetValueLen( buffer )`

  Returns byte length (see "tBMSLength" on page 2777) of the last value encoded to the buffer or decoded from the buffer.

- `BufGetValueBitLen( buffer )`

  Returns bit length (see "tBMSLength" on page 2777) of the last value encoded to the buffer or decoded from the buffer.

- `BufGetMode( buffer )`

  Returns buffer mode, see "tBMSMode" on page 2778.

- `BufInReadMode( buffer )`

  This is a boolean function returning `true` if the buffer is in the read mode (see "tBMSMode" on page 2778).

- `BufInWriteMode( buffer )`

  This is a boolean function returning `true` if the buffer is in the write mode (see "tBMSMode" on page 2778).

- `BufInNoMode( buffer )`

  This is a boolean function returning `true` if the mode of the buffer is not defined.

- `BufSetRule( buffer, rule )`

  This function sets the encoding rules to the buffer. Each setting overrides the previous. They are checked later in the encode and decode functions. The second parameter is combination of encoding rule masks of type `tERRule` (see "tERRule" on page 2778), it is automatically cast to `unsigned long`, which is the type of the second parameter. For example, PER without end padding encoding rule is set with an expression `BufSetRule(er_PER | er_NoEndPad)`. If not a complete set of rules is specified then `BufSetRule` will use default settings, for example `BufSetRule(er_PER)` is used (without bit masks `er_Aligned`, `er_Unaligned` and `er_NoEndPad`), then default PER variant will be applied which is configured by compilation switches, see "Compilation switches" on page 2817. The BufGetRule will return the encoding rules including those that have been applied depending on default settings.

- `BufGetRule( buffer )`

  This function returns value of type `unsigned long` which can be checked by binary "AND" operation with bit masks of type `ERRule` (see "tERRule" on page 2778), for example, `(SpecifiedRule & er_PER)` returns true if specified rule is PER.

Caution! Checking specified encoding rules

`unsigned long SpecifiedRule = BufGetRule( buffer );`

`( SpecifiedRule & er_PER & er_Aligned )` is not a correct check for PER Aligned.

`( SpecifiedRule & ( er_PER | er_Aligned ) )` should be used instead.

- `BufGetUserData( buffer )`

  This function returns a reference to the user data defined by `tUserData` type (see "User Data" on page 2810).

- `BufSetErrInitFunc( buffer, func )`

  This function sets a reference to the user error handling function. The interface is available only if `CODER_REMOVE_PATH` compile switch is absent (see "Error handling configuration" on page 2824).

- `BufGetErrInitFunc( buffer )`

  This function returns a reference to the user error handling function. `NULL` is returned if the function has not been set. The interface is available only if `CODER_REMOVE_PATH` compile switch is absent (see "Error handling configuration" on page 2824).

- `BufGetErrorPath( buffer )`

  This function returns a reference to the `tErrorPath` structure that contains a path of fields from the root type up to the field where an error has occurred. The C-representation of the `tErrorPath` is:

```
typedef struct
```

```
{
  void*          Fields[CODER_PATH_DEEP];
  unsigned long  NumOfFields;
  ...
} tErrorPath;
```

where `CODER_PATH_DEEP` is the maximum number of nested types, see “Error handling configuration” on page 2824. It is used within the user error handling function. The interface is available only if `CODER_REMOVE_PATH` compile switch is absent (see “Error handling configuration” on page 2824). The first element in the `Fields` array is a reference to the root Type Info - `tASN1TypeInfo*`. The next element might be one of the following:

`tASN1Component*` - for the SEQUENCE/SET component

`tASN1Alternative*` - for the CHOICE alternative

`tASN1Object*` - for an open object

`long*` - for the SEQUENCE OF/SET OF item index

- `BufGetMainVal( buffer )`

  This function returns a reference to the encode/decode value and it is used in the user error handling to assign the default contents to the value if an error has occurred. The interface is available only if `CODER_REMOVE_PATH` compile switch is absent(see “Error handling configuration” on page 2824).

- `BufSetErrorCode( buffer, error_code )`

  This function sets an error code (see “Error codes” on page 2799) to the buffer.

- `BufGetErrorCode( buffer )`

  This function returns an error code (see “Error codes” on page 2799).

- `BufGetByte( buffer )`

  Reads one byte from the buffer and returns it in a variable of type `unsigned char` after logically removing it from the buffer. The byte that has been returned will not be read from the buffer once again by another reading procedure. This function returns `unsigned char`.

- `BufPeekByte( buffer )`

Takes one byte from the buffer and returns it in a variable of type `unsigned char` without removing it from the buffer. The byte that has been returned can be peeked from the buffer any number of times. This function returns `unsigned char`.

- `BufPutByte( buffer, byte )`

  Puts one byte from the second parameter of type `unsigned char` to the buffer.

- `BufGetSeg( buffer, length )`

  Reads a segment of length (see "tBMSLength" on page 2777) bytes from a buffer and returns a segment pointer that points at the start of the retrieved segment. Note that this pointer should not be freed, because it is pointing into the memory segment of the buffer. The segment that has been returned will not be read from the buffer once again by another reading procedure. This function returns `unsigned char*`.

- `BufSkipSeg( buffer, length )`

  Skips `length` (see "tBMSLength" on page 2777) bytes from the buffer.

- `BufPeekSeg( buffer, length )`

  Takes `length` (see "tBMSLength" on page 2777) bytes from the buffer and returns a segment pointer from buffer without moving current pointer in a variable of type `unsigned char*`. The segment that has been returned can be peeked from the buffer any number of times. Note that this pointer should not be freed, because it is pointing into the memory segment of the buffer.

- `BufPutSeg( buffer, segment, length )`

  Puts a whole segment of `length` (see "tBMSLength" on page 2777) bytes to the buffer. The second parameter `segment` is of type `unsigned char*` and is a pointer to the segment start.

- `BufGetBit( buffer )`

  Reads one bit from the buffer and returns it in a variable of type `unsigned char` after logically removing it from the buffer. The bit that has been returned will not be read from the buffer once again by another reading procedure. This function returns `unsigned char`, where returned bit is the right-most bit in the `unsigned char` value.

- BufPutBit(buffer, bit)

  Puts one bit which is the right-most bit of function parameter of type unsigned char to the buffer.

- BufGetBits(buffer, number)

  Reads number bits from the buffer and returns them in number right-most bits in a variable of type unsigned long after logically removing them from the buffer. The bits that have been returned will not be read from the buffer once again by another reading procedure. The second parameter number is of type unsigned char. Bits are returned in a variable of type unsigned long, so number should be less or equal to sizeof(unsigned long)*8.

- BufPutBits(buffer, bits, number)

  Puts number right-most bits from a variable of type unsigned long to the buffer. Bits are passed in a variable of type unsigned long, so number should be <= sizeof(unsigned long)*8.

- BufAlign(buffer)

  This procedure skips all the bits from the buffer till the end of the byte.

- BufSetAppendBufFunc(buffer, func)

  Sets a reference to an append function. The append function is called when there are not enough bytes left in the buffer for a get operation and more bytes must be put into the buffer. Default is no append function.

Example 460: Sending a buffer

```
unsigned char * data;
unsigned int    datalen;

BufInitReadMode(buf);
datalen = BufGetDataLen(buf);
data = BufGetSeg(buf,datalen);
send_protocol(sa,data,datalen);
BufCloseReadMode(buf);
```

Example 461: Receiving from protocol and putting in a buffer

```
unsigned char data[MAXSIZE];
unsigned int datalen;
```

```
BufInitWriteMode(buf);
datalen=1;
while(datalen>0) {
   datalen = receive_protocol(sa,data,MAXSIZE,0);
   if (datalen>0)
      BufPutSeg(buf,data,datalen);
}
BufCloseWriteMode(buf);
```

You can write an append function that will be called when there are not enough bytes left in the buffer for a read operation. An example of when this can occur is during the execution of a decode function. The calling function calculates how many bytes that must be added to the buffer. The append function can choose to add more bytes than this. An append function can be called several times during a decode execution.

The buffer is in write mode when the append function is called and will automatically be set to read mode after the append function is finished. Do not use the `BufInitWriteMode` or `BufInitReadMode` macros in the append function.

If you want to add an append function for a buffer, then do the following steps:

1. Implement a C-procedure for appending data to the buffer. The C-procedure must have parameters compatible with `tBufAppendFunc`.

   ```
   typedef void (*tBufAppendBufFunc)(tBuffer buf,
   unsigned int len);
   ```
2. Use the macro `BufSetAppendBufFunc`, with the buffer and the append procedure as input parameters, immediately after a call to `BufInitBuf`.

Example 462: Buffer Append Procedure

```
void MyAppendBuf( tBuffer buf,
                  unsigned int minbytes );
{
    /* receive at least minbytes from
       protocol or sockets or ... */

    BufPutSeg(buf,seg,len );
}
```

Setting appending function:

```
BufInitBuf(buf);
```

```
BufSetAppendBufFunc(buf,MyAppendBuf);
```

- `BufGetAppendBufFunc( buffer )`

  Returns a reference to an append function. Returns `NULL` if the append function does not set.

Small Buffer Implementation

A small buffer has a memory segment, an array of `unsigned char`. The encoded bit patterns are written to this segment and the bit patterns to decode are read from it.

The buffer pointer is a pointer to a control structure and a data structure. The control structure and the data structure contains information and memory pointers that the small buffer uses.

In the data structure, there is a pointer that points at the beginning of the memory segment, a pointer that points at the end of the memory segment and a pointer that points at current position for a read or write operation.

Memory is allocated inside the small buffer implementation. An initial memory segment is allocated. The size of the initial segment is set by defining the macro `CODER_SMALLBUF_SIZE`, which has the default value equal to 0x1000, see "Buffers configuration" on page 2822. When a memory segment is full and a bigger segment is needed, then a 2 times larger segment is allocated, the contents in the old segment copied and the old segment freed. The macros `CUCF_ALLOC` and `CUCF_FREE` are used for all memory allocation and de-allocation.

SDL Interface to Buffer Management System

SDL CoderBuf interface

In the Extended SDL Interface, encoded data is stored in the C-buffer interface. From SDL, this data is accessed via the SDL type CoderBuf. The interface is not a direct mapping of the C-interface. Some of the functions are not accessible from SDL. For more detailed information about C-interface to the buffer management system see "C interface to buffer management system" on page 2777.

Below is a summary of the contents in the file `CoderBuf.sdl`.

Types

The SDL buffer interface type names are the same as the ones in the C buffer interface (see "Types and definitions" on page 2777). The following type is introduced in the SDL buffer interface

- Type `ptr_tBMSUserMemory` is a pointer to `tBMSUserMemory`.

Operators

- `BufInitBuf : CoderBuf, tBMSBufType [-> int];`
- `BufInitBufWithMemory : CoderBuf, tBMSBufType, ptr_tBMSUserMemory [-> int];`
- `BufCloseBuf : CoderBuf [-> int];`
- `BufCloseBufToMemory : CoderBuf, ptr_tBMSUserMemory [-> int];`
- `BufInitReadMode : CoderBuf [-> int];`
- `BufInitWriteMode : CoderBuf [-> int];`
- `BufCloseReadMode : CoderBuf [-> int];`
- `BufCloseDeleteReadMode : CoderBuf [-> int];`
- `BufCloseWriteMode : CoderBuf [-> int];`
- `BufGetDataLen : CoderBuf -> tBMSLength;`
- `BufGetReadDataLen : CoderBuf -> tBMSLength;`
- `BufGetByte : CoderBuf -> unsigned_char;`
- `BufPeekByte : CoderBuf -> unsigned_char;`
- `BufPutByte : CoderBuf, unsigned_char;`
- `BufSetRule : CoderBuf, tERRule;`
- `BufGetRule : CoderBuf -> tERRule;`
- `BufCopyBuf : CoderBuf, CoderBuf [-> int];`
- `BufPutSeg : CoderBuf, Octet_string, int;`
- `BufGetSeg : CoderBuf, int -> Octet_string;`
- `BufPeekSeg : CoderBuf, int -> Octet_string;`

These operators are mapped to the C buffer interface functions, so the semantics of SDL operators is the same as described for the corresponding C procedures in "Buffer Management Functions" on page 2779.

SDL CustomCoderBuf interface

If you want a solution of your own for accessing the coder library from within SDL, you can use `CustomCoderBuf`. This can be achieved by modifying the file `CustomCoderBuf.sdl` and writing two C-procedures. `CustomCoderBuf` can then be used in the calls of `encode` and `decode` instead of `CoderBuf`.

If only minor changes are of interest, you could copy the `CoderBuf.sdl` file to `CustomCoderBuf.sdl` as a start. Even when the goal is to make a completely new access interface, `CoderBuf.sdl` could be used as an example.

Implementing the procedures `ASN1EncodeCustomBuf` and `ASN1DecodeCustomBuf` in a file of your own, means that the file should then be included in the build process. For further information about parameters and return values of those procedures see the file `cucf_er_sdt.h` in the coder library. When `CustomCoderBuf` is selected in the user interface, the code generator generates code that calls those two procedures.

The procedures `ASN1EncodeCoderBuf/ASN1DecodeCoderBuf` and `ASN1EncodeOctet/ASN1DecodeOctet` could be seen as Telelogic Tau versions of a CustomCoderBuf. The first parameter in that implementation is `CoderBuf` and `Octet_string` respectively. When implementing your own `CustomCoderBuf`, you can select whatever type of this first parameter you want. However, it is important that the type defined in `CustomCoderBuf.sdl` file corresponds to the first parameter of the two above procedures.

The choice of using a custom implementation must be selected in the Analyzer dialog. See “Analyze SDL” on page 112 in chapter 2, *The Organizer*.

User defined Buffer Handling

The coder library can be configured to work with user defined buffers. It is possible to write your own specific buffer handling procedures which will be invoked instead of the default buffer handlers.

All coder library buffer implementations are based on the open buffer interface approach which allows you to introduce user types of buffer management in the same style as, for example, predefined small buffer management implementation.

If you want to use your own buffers, then perform the following steps:

1. Define the structure UserBufFuncs of type tBaseBuf (see "Types and definitions" on page 2777).

2. Set the value bms_UserBuffer to the field BufType.

3. Implement the buffer access functions and assign references to these functions to the corresponding fields of UserBufferFuncs. Buffer access functions should have compatible interfaces with the function pointer types defined for tBaseBuf structure in the file bms.h (see "Buffer Management System sub-directory" on page 2815) and should support the semantics of buffer access functions described in "Buffer Management Functions" on page 2779.

Note: Buffer implementation example

The small buffer implementation is based on the open buffer interface and can be used as an example of a user defined buffer implementation. Looking into the files bms_small.h and bms_small.c in the installation (see "Buffer Management System sub-directory" on page 2815) will be helpful before starting the implementation.

4. Compile coder library with CODER_USE_USERBUF compile switch, see "Buffers configuration" on page 2822,

5. Compile and link the application with the user buffer handling functions and the defined structures.

Memory Management System

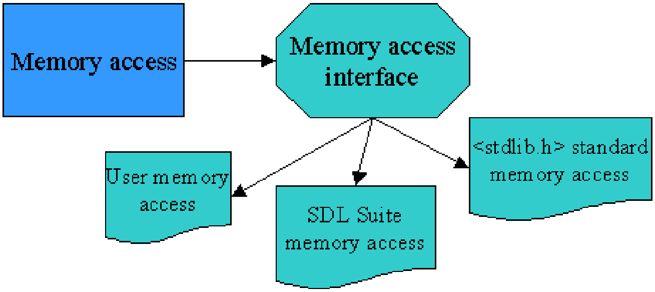

Figure 501: Memory management

The coder solution contains memory management functions. There are only two basic functions used for memory management: alloc and free.

Memory management in the coder library consists of allocation and freeing of memory, and releasing of memory in case of errors.

Coder library contains two levels of memory access functions:

- Pure allocation and freeing of memory - `CUCF_ALLOC(size, type)` and `CUCF_FREE(ptr, size, type)`,
  - where `size` is of type `size_t` and represents the size of memory in bytes to be allocated,
  - `type` is not used (reserved for the future),
  - `ptr` is of type `void*` and is a pointer to the memory to be released.
- Safe memory allocating and releasing - `CUCFAlloc(info, size, type)`, `CUCFFree(info, ptr, size, type)` and `CUCFRelease(info, ptr)`, where `size`, `type` and `ptr` are the same as for the previous memory handling functions. Apart from allocating, `CUCFAlloc` function saves a pointer to the allocated memory block in the internal stack, which is used for memory freeing in case of error. The first parameter `info` in these three memory handling functions is of type `tEMSInfo*` and it is a pointer to the stack where all

allocated memory segments are stored. Safe functions use pure memory management functions for memory operations. `CUCFAlloc` calls `CUCF_ALLOC` for allocating memory for storage cell in the stack and for allocating `size` required bytes, `CUCFFree` frees memory pointed by `ptr` and also the corresponding stack cell.

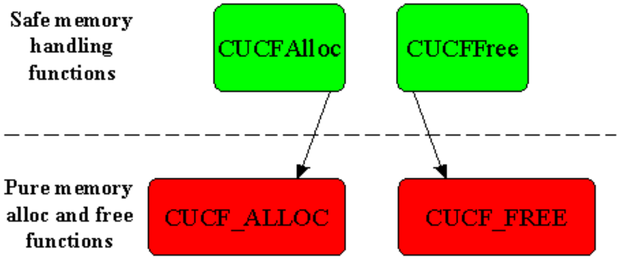
Safe memory handling functions
CUCFAlloc
CUCFFree
Pure memory alloc and free functions
CUCF_ALLOC
CUCF_FREE

Figure 502: Memory management functions

Safe memory handling functions are used in those places where memory freeing procedure will not be performed in case of error, for example, in the coder library kernel. Pure memory handling functions are called from the safe functions. They are also called directly in those places where memory can still be released even after an error has occurred, for example, buffer memory allocation (memory is released by BufCloseBuf function).

Predefined Memory Handling

There are two types of predefined memory handling functions which can be used to perform allocation and freeing of memory:

- SDL Suite memory handlers `XALLOC` and `XFREE`, see "Functions for Allocation and Deallocation" on page 3039 in chapter 62, *The Master Library*, and
- standard `malloc` and `free` functions from `<stdlib.h>` which are the default memory handlers.

Users can also define their own application specific memory handling functions, see "User defined Memory Handling" on page 2794.

The type of memory handler to be invoked for allocation and freeing of memory is configured when compiling coder library by compile switches, see "Memory management configuration" on page 2823.

User defined Memory Handling

The coder library can be configured to work with user-specific memory handling procedures. It is possible to write your own allocate and free procedures, which will be invoked instead of the default memory handlers.

> **Note: Performance optimization**
>
> Fast memory handling, if applicable, can help optimize the performance of the executable quite a lot. One of the examples of fast memory handling is working with statically allocated memory instead of dynamically allocated in alloc and free procedures.

If you want to use your own memory handler, perform the following steps:

1. Create file `mms_user.h` and insert definitions of macros `USER_ALLOC_FUNC(size, type)` and `USER_FREE_FUNC(ptr, size, type)` to point to the user defined memory handling procedures. These macros will be used for redefining pure memory handling procedures.
2. Compile the coder library with the `CODER_MMS_USER` compile switch, see "Memory management configuration" on page 2823.
3. Add include path for the file `mms_user.h` to the compilation settings.
4. Compile and link the user memory handler function to the application.

Example 463: File mms_user.h ––––––––––––––––––––––––––––––––

```
#ifndef mms_user_h
#define mms_user_h

#define USER_ALLOC_FUNC(size,type)  UserAlloc(size)
#define USER_FREE_FUNC(ptr,size,type) UserFree(ptr)

extern void* UserAlloc(size_t size);
extern void UserFree(void* ptr);
```

```
#endif
```

There are also two BMS user-specific memory handling procedures that are referenced through macros defined in the mms_user.h:

```
#define USER_BMS_ALLOC_FUNC(buffer, size, type)
UserBmsAlloc(buffer, size)
#define USER_BMS_FREE_FUNC(coder, ptr, size, type)
UserBmsFree(buffer, ptr, size)
```

These interfaces are used only in the buffer functions (see "Buffer Management Functions" on page 2779). By default (without this definition) USER_ALLOC_FUNC and USER_FREE_FUNC are used inside the buffer functions.

The BMS user-specific memory handlers use buffer access types tCoder* or tBuffer as the first argument. This argument opens the access to the tUserData (see "User Data" on page 2810) inside the BMS memory handling:

```
void* UserBmsAlloc(tCoder* Coder, size_t Size)
{
  tUserData* data = BufGetUserData(Coder);
  ... /* user implementation */
}

void UserBmsFree(tCoder* Coder, void* Ptr, size_t
Size)
{
  tUserData* data = BufGetUserData(Coder);
  ... /* user implementation */
}
```

Error Management System

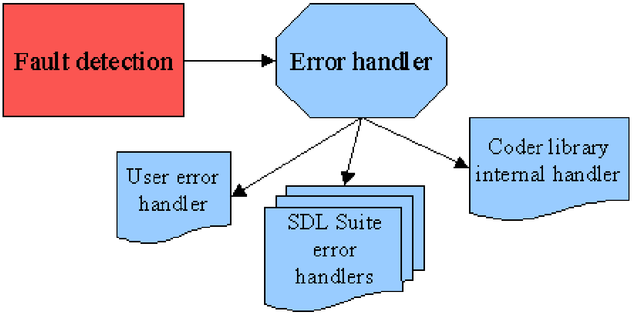

Figure 503: Error management

The coder solution contains error management functions. The error management consists of fault detection, error handling and error output. The error management is optional and can be removed from the application.

The fault detection consists of monitoring code that checks for all kinds of faults when encoding and decoding. When a problem is detected the fault detection part will start the error handler. The monitoring code is controlled by C macros and can be removed by defining macros, see "Error checking configuration" on page 2819. The user defined error handling functionality is described in "User defined Error Handling" on page 2806.

SDL Suite error output functionality has several error output implementations for the different types of kernels. The behavior is different for simulation and for application. It is possible to register a different error output function. See "User defined Error Output" on page 2809.

There are several types of error output functions which can be used to perform error output:

- SDL Suite error output as default
- Coder library error output
- user defined application specific error output functions.

The type of error output to be invoked upon error detection is configured when compiling coder library by compile switches, see "Error output configuration" on page 2824.

Error Management Interface

Encode and decode functions return a status code, which is equal to a success code, or one of the error codes in case of failure. Error codes are visible in both SDL and C functional interfaces. If an error occurs, its code is returned by the encoding or decoding function, and is written into the buffer at the same time.

The behavior of the fault detection can be controlled by compilation switches, that are described in "Error checking configuration" on page 2819. For instance, it is possible to remove all fault detection by defining a corresponding switch.

Detailed error messages

Error messages contain a type name with a path to the field where an error has occurred. The path contains a sequence of SET/CHOICE/SEQUENCE fields and open type objects from the root type up to the field where an error has occurred.

There are two possible representations for the path:

- Path fields are represented by their names. This representation is available only if one of the following compile switches are present, `CODER_TI_NAMES` (see "Error output configuration" on page 2824) or `CODER_VMS_PRINT` (see "Printing configuration" on page 2825). For example:

  ```
  ERROR 21 No more data for read at BOOLEAN, type:
  BOOLEAN, path: yASN1_Type1.field2.comp1
  ```
- Path fields are represented by their indexes - in this case root type is named "undef" and path contains indexes of SEQUENCE/SET components, CHOICE alternatives and open type objects, for example:

  ```
  ERROR 21 No more data for read at BOOLEAN, type:
  undef, path: SEQUENCE.2.1
  ```

By default the second representation with indexes is used.

If an error occurs within the encoding/decoding then you can unambiguously understand the exact place of the error. For example if we have a complex type with the several nested sequences:

```
Type1 ::= SEQUENCE {
  field1 Type2,
  field2 Type2
}
Type2 ::= SEQUENCE {
  comp1 [0] Bool,
  comp2 [1] Int
}
Bool ::= [0] BOOLEAN
Int ::= [1] INTEGER
```

And the error is:

```
ERROR 21 No more data for read at BOOLEAN, type:
BOOLEAN, path: yASN1_Type1.field2.comp1
```

(in the first representation)

```
ERROR 21 No more data for read at BOOLEAN, type:
undef, path: SEQUENCE.2.1
```

(in the second representation)

Then the exact place of the error is known. The error occurred in `Bool` type in the second nested sequence by the path `Type1.field2.comp2` .

The benefits of this feature become significant in a huge ASN.1 specifications with the long nested fields, like UMTS. This is an example of a real error message from the RRC specification (subpart of UMTS):

```
ERROR 22: Wrong TAGGED identifier octet, type:
CPCH_SetInfo_INLINE_3, path: yASN1_DL_DCCH_Message.mes-
sage.radioBearerSetup.modeSpecificPhysChIn-
fo.fdd.cpch_SetInfo.ap_AICH_ScramblingCode
```

Detailed error messages (type and path) is used in the `ec_VAL_` and `ec_DEC_` errors.

The default configuration includes detailed error messages. Detailed messages require extra resources from the speed and from the Type Info size (if names information is enabled). The encoding/decoding speed becomes a bit slower because the feature requires extra manipulations with the stack to store the temporary path. If you do not need to use this feature then you should use `CODER_REMOVE_PATH` compile switch (see "Error handling configuration" on page 2824). Error messages in this case will look like:

```
ERROR 21 No more data for read at BOOLEAN, type:
undef, path: undef
```

SDL error interface

In SDL, error codes are represented by integer values and are defined as Integer synonyms. Names of synonyms are the same as error codes described in "Error codes" on page 2799. The file codererrors.sdl contains a complete list of Integer synonyms for coder error codes. This file must be included to the SDL system that is going to check coder error codes after encoding and decoding.

C error interface

In the C error interface, error codes are literals of one big enumerated type, and they can be treated as C integer values. All possible error codes are listed in the file errors.h in the 'ems' folder of the library installation (see "Files and File Descriptions" on page 2812). For detailed description of error codes see "Error codes" on page 2799.

Error codes

If the decode or encode succeeded, ec_SUCCESS is returned.

In case of failure, an error code is returned that helps to understand the reason of the failure. All possible error codes with short descriptions are listed below.

Memory access errors

ec_MEM_NotEnoughMemory

The application has run out of memory and one of the coder library functions can not allocate memory.

Buffer errors

ec_BUF_DifferentBufferTypes

This is a buffer copying error. The BufCopyBuf function belongs to the general buffer interface which supports different buffer types. The error occurs when the source and destination buffers in the copying function are of different buffer types.

ec_BUF_WrongBufferType

The message is reported when the buffer initialize function has been called with the buffer type which has been excluded from the application by compile switches (see "Buffers configuration" on page 2822).

ec_BUF_CloseNotInitialized

The message is reported when `BufCloseBuf` is called on a buffer that has not been initialized.

ec_BUF_WorkWithNotInitialized

One of the two buffer modes, read or write, has to be initialized before accessing the buffer with reading or writing procedures. If a read or write procedure has been applied to a buffer that has not been initialized, the ec_BUF_WorkWithNoneInitialized error is reported.

ec_BUF_NullPtrToUserMemory

BufInitBufWithMemory or BufCloseBufToMemory is called with `NULL` pointer to user memory.

ec_BUF_NotEnoughUserMemory

This error is specific for buffers that are initialized by the user memory, and it denotes that the size of the user memory assigned to the buffer is not enough for storing data.

ec_BUF_OpenOpened

This error is reported when trying to reopen the buffer which has already has been opened for write or read mode, and has not been closed afterwards.

ec_BUF_IllegalClose

This error is reported when BufCloseBuf is called on the buffer for which one of the modes, reading or writing, has not been closed.

ec_BUF_CloseWrongMode

This error is returned when trying to close the buffer for a mode different from the opened one. This situation can happen, for example, when BufCloseWriteMode follows the function BufInitReadMode, or when

BufCloseReadMode is called to the buffer which mode has been initialized by the function BufInitWriteMode.

ec_BUF_OperationWrongMode

This error denotes that you are trying to perform an incorrect operation for the current buffer mode. For example, you are trying to read data from the buffer which has been opened for writing by function BufInitWriteMode.

ec_BUF_NoMoreDataForRead

This error is returned when you are trying to read data from a buffer which is already empty.

ec_BUF_TooBigNumberOfBits

The error is returned when you are trying to put or get more bits from the buffer than it is possible to pass to the bit buffer interface function. The operations BufGetBits and BufPutBits operate with bits stored in the value of type long and can not process more then sizeof(long) number of bits. So when they are called with a number of bits more then sizeof(long), an error message is reported.

ec_BUF_InvalidEncodingRules

This error code is returned when the combination of bit masks passed as the second parameter to the function BufSetRule does not form a valid encoding rule. For example, `BufSetRule( buffer, er_BER | er_Aligned )` will return this error code because `er_Aligned` is an alternative of PER and does not have any sense together with BER.

Value errors

ec_VAL_IllegalRealBase

The real value received for encoding is represented with a base that cannot be handled by the encoder function.

ec_VAL_WrongConstrainedValue

The value passed to the coder function does not satisfy constraints specified for the corresponding type.

ec_VAL_WrongConstrainedLength

The size of the value passed to the coder function does not satisfy the size constraint specified for the corresponding type.

ec_VAL_WrongConstrainedAlphabet

Characters in the string value passed to the coder function do not satisfy the permitted alphabet constraint specified for the corresponding type.

Decoding errors

ec_DEC_NoMoreDataForRead

This error is returned when you are trying to read data from a buffer which is already empty. There is a difference in the formats between `ec_DEC_NoMoreDataForRead` and `ec_DEC_NoMoreDataForRead`. `ec_DEC_NoMoreDataForRead` uses the detailed error message approach (with type and path) but `ec_BUF_NoMoreDataForRead` does not.

ec_DEC_WrongIdentifierOctet

This is a BER specific error, based on BER encoding TLV (Type-Length-Value) structure. The type information from TLV is stored in a so called identifier octet. If the data in the identifier octet for the sequence of bits to be decoded is wrong (for example, ASN.1 type is defined to be INTEGER, but the identifier octet for the value of that type claims that value to be decoded if of type BOOLEAN), an error is returned.

ec_DEC_WrongLength

For some ASN.1 types, the range of the encoded value length is restricted. When length is out of range, the wrong length error is returned.

ec_DEC_TooBigTagNumber

Tag numbers are stored in the variable of type unsigned long, so the value of a tag is restricted. When the tag number occupies more then 32 bits, restriction is violated and this error message is returned.

ec_DEC_WrongConstructedLengthPrefix

When constructed encoding is used, each group of bytes from the encoding is prefixed by the group information. For example, if it is the last group in the constructed encoding or not, and similar. Constructed encoding prefix occupies 8 bits, but not all bit combinations are meaningful. When the combination is wrong, this error is returned.

ec_DEC_WrongUnusedBits

This is a BER specific error. BIT STRING values in BER are encoded into a sequence of bytes, although they cannot contain exactly an integer number of 8 bits. For correct value encoding, the number of unused bits in the last byte is also encoded. When this number is not in the range between 0 and 7, an error is returned.

ec_DEC_TooBigSubIdent

This object identifier decoding error is reported when the sub-identifier is too big and cannot be stored in the variable of type long (4 bytes).

ec_DEC_WrongRealPrefix

The real value encoding contains a prefix denoting the type of real value that has been encoded, plus infinity and minus infinity real values, zero real value, other common real value (mantissa + base + exponent). The real prefix is encoded into 8 bits but not all of them have meaning. If the real encoding contains a bad prefix, this error is reported.

ec_DEC_UnsupportedRealBase

Trying to decode a real value which has been encoded with a base that is not supported in the coding library.

ec_DEC_UnsupportedRealDecimalEncoding

Decimal encoding of real values is not supported. If real value has been encoded according to decimal real encoding algorithm, the decoder will not be able to decode it and it will return this error code.

ec_DEC_RequiredComponentIsAbsent

This error can be reported when decoding SEQUENCE or SET type values and it denotes that not all required components are present in the value encoding.

ec_DEC_ExtRequiredComponentIsAbsent

This is SEQUENCE and SET decoding specific error. It points out that not all required components from extension addition group are present in the value encoding, although the group itself is encoded as present.

ec_DEC_AbsentComponentIsPresent

This error can appear while decoding a value of sequence or set type with the ABSENT constraint applied. When the component constrained to be absent is present in the encoded value, this error message is reported.

ec_DEC_AbsentAlternativeIsPresent

This error can appear while decoding a value of choice type with the ABSENT constraint applied. When the alternative constrained to be absent is present in the encoded value, this error message is reported.

ec_DEC_UnknownComponent

This is SEQUENCE and SET specific decoding error, and it is reported when the encoded field tag is not a known tag from the type fields of SEQUENCE or SET type.

ec_DEC_UnknownAlternative

This is CHOICE specific error, and it is returned when the encoded choice alternative has got a tag which does not belong to the set of possible tags for the decoded choice type.

ec_DEC_NoOpenId

This is open type specific error. It is reported when open type value can not be decoded because open type identifier field is absent for some reason in the encoding.

ec_DEC_UnknownObject

This is open type specific error. It is reported when decoded open type identifier is not equal to any of identifier in the open type restricting table.

ASCII decoding errors

Each ASCII decoding error refer to a particular type and denotes that there are problems or errors when decoding value of that type, for example, ec_ERROR_DECODING_PARSTART is reported when there are errors when decoding start of signal parameter or variable from the buffer.

All possible ASCII decoding errors are listed below.

ec_ERROR_DECODING_INTEGER
ec_ERROR_DECODING_REAL
ec_ERROR_DECODING_BOOLEAN
ec_ERROR_DECODING_TIME
ec_ERROR_DECODING_CHARSTRING
ec_ERROR_DECODING_BIT
ec_ERROR_DECODING_BITSTRING
ec_ERROR_DECODING_OCTET
ec_ERROR_DECODING_OCTETSTRING
ec_ERROR_DECODING_SIGNALID
ec_ERROR_DECODING_STRUCT
ec_ERROR_DECODING_CHOICE
ec_ERROR_DECODING_POWERSET
ec_ERROR_DECODING_BAG
ec_ERROR_DECODING_STRING
ec_ERROR_DECODING_ARRAY
ec_ERROR_DECODING_REF
ec_ERROR_DECODING_USERDEF
ec_ERROR_UNKNOWN_TYPE_NODE
ec_ERROR_DECODING_PARSTART
ec_ERROR_DECODING_PAREND

Internal errors

ec_INT_InternalError

This is coder library internal error. If coder function returns this error, please, contact Telelogic Customer Support. Internal error will never be reported if `CODER_CHECK_NONE_INNER` compile switch is set up (see “Error checking configuration” on page 2819), in this case all internal error checks are removed from the library code.

ec_INT_UnsupportedType

This is also coder library internal error. It is reported when the type specified for the encode or decode function in the ASN.1 type info structure is not supported by the coder library. Internal error will never be reported if `CODER_CHECK_NONE_INNER` compile switch is set up (see "Error checking configuration" on page 2819), in this case all internal error checks are removed from the library code.

> **Note: Error codes backwards compatibility**
>
> Backwards compatibility is implemented for those error codes which have been removed or renamed in comparison with the previous versions of the coder library error management system. All old error names will be mapped to new ones with the same functionality for the systems using previous version of coder library. An example of error code which is not used in the new coder library any more is `ec_SMLBUF_READ_ERROR`.

Definition of your own error codes

It is possible to specify your own error codes. It might be useful for user encoding rules or user memory management implementation.

If you want to use your own error codes, perform the following steps:

1. Create file `errors_user.h` and insert your own error codes by using the following format:

CUCF_ERROR(ec_<error identifier>, "Error message")

Example 464: File errors_user.h ––––––––––––––––––––––––––––––––

```
CUCF_ERROR(ec_MY_ERROR_ONE, "My error one")
CUCF_ERROR(ec_ME_ERROR_TWO, "My error two")
```

––

2. Compile the coder library with the `CODER_EC_USER`=”errors_user.h”.
3. Add include path for the file `errors_user.h` to the compilation settings.

User defined Error Handling

This functionality allows to call a user defined error handling function inside the decoding procedure. If an error occurs this function will be called. Combined with the `BufGetErrorPath( buffer )` buffer interface (see "Buffer Management Functions" on page 2779), error handling function gives an opportunity to insert a default value to the field for which the decoding failed.

An error handler function must be written by the user according to the following format:

```
typedef void (*tBufErrInitFunc)(tBuffer Buffer);
```

There is an interface in the buffer functionality `BufSetErrInitFunc( buffer, func )` that allows to set user error handling by reference into the buffer and it will be used by the decoder. `BufGetErrInitFunc( buffer )` returns a reference to an error handling function. By default there is no error handler and it returns `NULL`. Below there is an example with the user defined error handling.

These two checks between ErrorPath and TypeInfo help to determine the exact place where an error has occurred:

```
#define EQ_TI(field, ti) \
  ((tASN1TypeInfo*)(field)==(tASN1TypeInfo*)&ti)
#define EQ_CMP(field, ti) \
  (((tASN1Component*)(field))->TypeInfo==(tASN1TypeInfo*)&ti)
```

Error handling:

```
void TinyErrorHandling(tBuffer b)
{
  int res = BufGetErrorCode(b);
  tErrorPath *ep = BufGetErrorPath(b);
  LineB* lineB = (LineB *)BufGetMainVal(b);
  if ( res == ec_VAL_WrongConstrainedValue &&
       ep->NumOfFields == 2 &&
       EQ_TI(ep->Fields[0], yASN1_LineB) &&
       EQ_CMP(ep->Fields[1], yASN1_PointB_INLINE_0))
    {
      lineB.point1.x = 40;
      return;
    }
  if ( res == ec_VAL_WrongConstrainedValue &&
       ep->NumOfFields == 2 &&
       EQ_TI(ep->Fields[0], yASN1_LineB) &&
       EQ_CMP(ep->Fields[1], yASN1_PointB_INLINE_1))
    {
      lineB.point2.y = 25;
      return;
    }
}
```

ASN.1:

```
Tiny DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
  PointA ::= SEQUENCE {
    x        INTEGER (0..42),
    y        INTEGER (0..26)
  }
  PointB ::= SEQUENCE {
    x        INTEGER (0..40), -- lineA.point2.x = 42
    y       INTEGER (0..25)  -- lineB.point1.y = 26
  }
  LineA ::= SEQUENCE {
    point1   PointA,
    point2   PointA
  }
  LineB ::= SEQUENCE {
    point1   PointB,
    point2   PointB
  }
  lineA LineA ::= { point1 { x 11, y 26 }, point2 {
x 42, y 24 } }
  lineB LineB ::= { point1 { x 11, y 25 }, point2 {
x 40, y 24 } }
END
```

The code example with the encoding of `LineA` type and `LineB` as the type for the decoding. The encoded `lineA` value does not satisfy the `LineB` constraint and that is why there are two constraint errors:

```
ERROR 18 Wrong INTEGER constrained value, type:
PointB_INLINE_0, path: yASN1_LineB.point1.x
ERROR 18 Wrong INTEGER constrained value, type:
PointB_INLINE_1, path: yASN1_LineB.point2.y
```

The `TinyErrorHandling` function catches these errors with a corresponding assignment to the correct values.

```
BufInitBuf(b, bms_SmallBuffer);
BufSetErrInitFunc(b, TinyErrorHandling); /* set
TinyErrorHandling as decoding
BufInitWriteMode(b);
res = ASN1_ENCODE(b, (tASN1TypeInfo *)&yASN1_LineA,
lineA);
BufCloseWriteMode(b);
BufInitReadMode(b);
res = ASN1_DECODE(b, (tASN1TypeInfo *)&yASN1_LineB,
dec_lineB); /*
BufCloseReadMode(b);
BufCloseBuf(b);
```

User error handling requires the same resources as for detailed error messages (see “Detailed error messages” on page 2797). The encoding/decoding speed becomes slower because of some extra manipulations with the stack. CODER_REMOVE_PATH compile switch (see "Error handling configuration" on page 2824) should be defined to disable the error handling.

User defined Error Output

It is possible to write your own error output function, which will be invoked when a fault is detected.

If you want to use your own error output, perform the following steps:

1. Implement a C-procedure for error output with the prototype void USERErrorOutputFunc(FILE* File, tEMSErrorCode Code, va_list MessageArguments). tEMSErrorCode is an enumerated type containing all possible error codes returned by coder library functions, see "Error codes" on page 2799. You can also find a list of the different error codes in errors.h from /cucf/ems folder.
2. Compile the coder library with CODER_EO_USER compile switch, see "Error output configuration" on page 2824,
3. Compile and link the user error output function to the application.

The USERErrorOutputFunc function takes error message arguments as the third parameter which corresponds to the message string defined in the coder library. The error management system provides a function that allows you to get the coder library message string for the error code: char* CUCFGetErrorMessage(tEMSErrorCode Code). Example 465 illustrates one possible way of defining a user error output function.

Example 465 Error output --

```
void USERErrorOutputFunc( FILE* File,
                          int Code,
                        va_list MessageArguments )
{
  /* user error output with the code and message */
  fprintf(File, "USER ERROR %d ", Code);
  vfprintf(File, ErrorMessageArray[Code],
MessageArguments);
  fputc('\n', File);
}
```

User Data

There is an option to include your own data fields into the buffer type. C-type tUserData is defined in the user_data.h file and it can be included into the buffer by CODER_USER_DATA compile switch (see "User data configuration" on page 2825). By default type tUserData is defined as void*, but you can redefine it according to your needs.

tUserData with your own fields might be useful in the user error handling function (see "User defined Error Handling" on page 2806) to store the data for the later manipulation outside the decoder or in the USER_BMS_ALLOC_FUNC/USER_BMS_FREE_FUNC (see "User defined Memory Handling" on page 2794). The buffer interface BufGetUserData(buffer) (see "Buffer Management Functions" on page 2779) returns a reference to tUserData for your further handling. If you want to use your own data fields as tUserData, perform the following steps:

1. Create file user_data.h and insert declaration of the tUserData C-structure.
2. Compile the coder library with the CODER_USER_DATA compile switch.
3. Add include path for the file user_data.h to the compilation settings.

Example 466: File user_data.h ––––––––––––––––––––––––––––––––

```
#ifndef user_data_h
#define user_data_h

typedef struct {
  int UserInt1;
  int UserInt2;
  int UserInt3;
} tUserData;

#endif
```

Printing Opportunities

The print functions can be used for test and debug purposes. They will be available at runtime only if the compilation switch CODER_VMS_PRINT is set, see "Printing configuration" on page 2825.

There are two print functions available, ASN1_PRINT_TYPE and ASN1_PRINT. The print functions use the type information in the same way as an encoding function.

The function ASN1_PRINT_TYPE prints the contents of the type information structure for one particular type. There are two input parameters, the first is a file handle and the second is a reference to the type information.

The function ASN1_PRINT prints the value of a variable or signal parameter. This print function has got three input parameters, the first is a file reference, the second is a reference to the type information structure and the third is a reference to the variable or the signal parameter.

Example 467: Print type information ———————————————————

```
FILE * logfile;

logfile = fopen( "asn1print.log", "w" );

ASN1_PRINT_TYPE( logfile,
                  (tASN1TypeInfo *)&yASN1_TestType);

fclose( logfile );
```

———————————————————————————————

Example 468: Print value of signal parameter ———————————————

```
FILE * logfile;

logfile = fopen( "asn1print.log", "w" );

ASN1_PRINT( logfile, (tASN1TypeInfo *)&yASN1_TestType,
     (void *)&((yPDef_outsig *)(*SignalOut))->Param1);
fclose( logfile );
```

———————————————————————————————

Structure and Configuration

An application with ASN.1 encoding or decoding support contains the following:

- Encoding and decoding functions. One set of functions for BER encoding and decoding and one set of functions for PER encoding and decoding.

> **Note: ASCII coder**
>
> ASCII encoding and decoding, which is based on Telelogic Tau SDL Suite internal encoding rules is also available. ASCII encoding and decoding is described in “SDL Data Encoding and Decoding, ASCII coder” on page 2717 in chapter 58, *Building an Application*.

- Buffer management functions.
- Error management functions (optional).
- Print functions (optional).
- ASN.1 type information generated by the ASN.1 Utilities.
- SDL type and operator information generated by the SDL C code generators.

Files and File Descriptions

This section describes the static file structure of the coder directories in the installation.

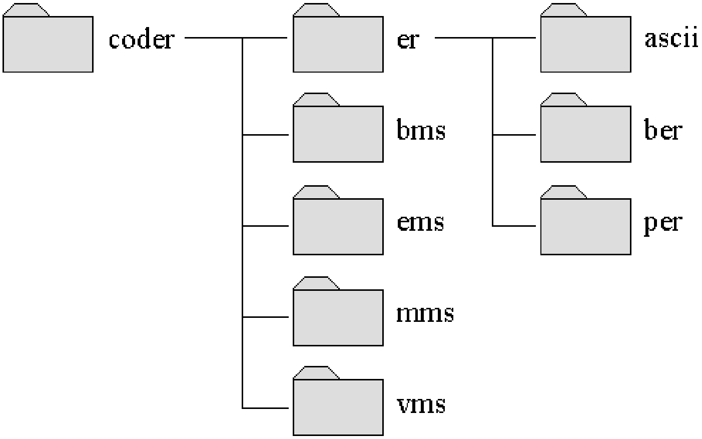

Figure 504: The coder directory

The coder directory contains five sub-directories which correspond to the functional modules in the coder library:

- `coder/er` - this directory contains three sub-directories which correspond to different encoding rules supported by the coder library.
- `coder/bms` - Buffer Management System files.
- `coder/ems` - Error Management System files.
- `coder/mms` - Memory Management System files.
- `coder/vms` - Value Management System files.

There are also several files included to the root coder directory:

- `cucf.h` - main header file of the coder library. It includes all useful header files from the library functional modules.
- `cucf_cfg.h` - standard C library includes, like `<stdio.h>`, `<sdtlib.h>` and others.
- `cucf_er.h` - declarations of main encode and decode coder interface functions.
- `cucf_er.c` - definition of main encode and decode coder interface functions.

- `cucf_er_sdt.h` - declarations of SDL Suite specific encode and decode interface functions which are used for encode and decode within SDL support.
- `cucf_er_sdt.c` - definition of SDL Suite specific encode and decode interface functions which are used for encode and decode within SDL support.
- `cucf_er_ttcn.h` - declarations of TTCN Suite specific encode and decode interface functions which are used for encode and decode within TTCN support.
- `cucf_er_ttcn.c` - definition of TTCN Suite specific encode and decode interface functions which are used for encode and decode within TTCN support.
- `coderascii.h`, `coderber.h`, `coderper.h`, `coderucf.h` - these files are not used inside the library, but provide backwards compatibility solutions for the SDL systems created with the previous versions of SDL coder support.

Encoding rules sub-directory

Sub-directory `coder/er` contains three sub-directories corresponding to different types of encoding rules supported by the coder library:

- `ascii`

  This directory contains files with encoding and decoding function according to internal rules of the SDL Suite. The ASCII coders are described in "SDL Data Encoding and Decoding, ASCII coder" on page 2717 in chapter 58, *Building an Application*.

- `ber`

  This directory contains files with encoding and decoding functions according to BER (Basic Encoding Rules), see "BER sub-directory" on page 2814.

- `per`

  This directory contains files with encoding and decoding functions according to PER (Basic Encoding Rules), see "PER sub-directory" on page 2815.

BER sub-directory

Sub-directory `coder/er/ber` contains the following files:

- `ber.h` - header file with declaration of `BEREncode` and `BERDecode` functions.

- ber_base.h - contains declarations of basic functions needed by the encode and decode functions.
- ber_base.c - contains implementation of basic functions needed by encode and decode functions.
- ber_content.h - contains declarations of functions that encode and decode ASN.1 predefined types.
- ber_content.c - contains implementation of functions that encode and decode ASN.1 predefined types.
- ber_decode.c - definition of main outermost function BERDecode which references functions from ber_base and ber_content modules.
- ber_encode.c - definition of main outermost function BEREncode which references functions from ber_base and ber_content modules.

PER sub-directory

Sub-directory coder/er/per contains the following files:

- per.h - header file with declaration of PEREncode and PERDecode functions.
- per_base.h - contains declarations of basic functions needed by the encode and decode functions.
- per_base.c - contains implementation of basic functions needed by encode and decode functions.
- per_content.h - contains declarations of functions that encode and decode ASN.1 predefined types.
- per_content.c - contains implementation of functions that encode and decode ASN.1 predefined types.
- per_decode.c - definition of main outermost function PERDecode which references functions from per_base and per_content modules.
- per_encode.c - definition of main outermost function PEREncode which references functions from per_base and per_content modules.

Buffer Management System sub-directory

Sub-directory coder/bms contains the following files:

- bms.h - base buffer management declarations, this header contains base buffer control structure definitions. This is the main header file for buffer management.

- `bms.c` - implementation of base buffer functions which are common for all types of buffers.
- `bms_small.h` - small buffer management specific declarations.
- `bms_small.c` - implementation of small buffer management.

Error Management System sub-directory

Sub-directory `coder/ems` contains the following files:

- `ems.h` - error management declarations. This is the main header file for coder error handling.
- `ems.c` - implementation of common error management functions.
- `ems_eo_sdt.h` - SDT specific error management declarations and includes.
- `ems_eo_sdt.c` - implementation of SDT specific error handling functions.
- `errors.h` - declaration of error codes and error information.

Memory Management System sub-directory

Sub-directory `coder/mms` contains the following files:

- `mms.h` - declarations of pure memory handling macros `CUCF_ALLOC` and `CUCF_FREE` and safe memory management functions `CUCFAlloc` and `CUCFFree` (see "Memory Management System" on page 2792). This is the main header file for coder memory handling.
- `mms.c` - implementation of safe memory handling functions.

Value Management System sub-directory

Sub-directory `coder/vms` contains the following files:

- `vms.h` - this is the main header file for value management system, it contains includes of all other useful header files from this folder.
- `vms_type.h` - definitions of types used in ASN.1 type nodes.
- `vms_macro.h` - definitions of macros which are used for declaration and filling ASN.1 type structure nodes. Calls to these macros are generated by ASN.1 coder generator.
- `vms_vr_sdt.h` - definitions of macros which are used for inserting SDL Suite specific information to ASN.1 type structure nodes. Calls to these macros are generated by ASN.1 coder generator only for SDL Suite coder generation.
- `vms_vr_ttcn.h` - definitions of macros which are used for inserting TTCN Suite specific information to ASN.1 type structure nodes.

Calls to these macros are generated by ASN.1 coder generator only for TTCN Suite coder generation.

- `vms_export.h` - type node declarations for predefined ASN.1 types, such as `BOOLEAN`, `INTEGER`, `BIT STRING`, `OCTET_STRING`, `NULL`, `OBJECT IDENTIFIER`, `REAL`, string types.
- `vms_export.c` - type node definitions for predefined ASN.1 types.
- `vms_internal.h` - internal value representation type and macro definitions.
- `vms_base.h` - declarations of internal value representation access procedures.
- `vms_base.c` - implementation of internal value representation access procedures.
- `vms_check.h` - declarations of check procedures.
- `vms_check.c` - implementation of procedures that perform error checking of type nodes and values.
- `vms_print.h` - declarations of print procedures used for test and debug purposes.
- `vms_print.c` - implementation of print procedures used for test and debug purposes.

Compilation switches

You can change the default properties and configure the BER and PER coders by setting the compilation switches. Some of these switches can be set by choosing options in the Targeting Expert, see chapter 60, *The Targeting Expert*.

Available compile switches can be separated onto several groups. Below there are descriptions of compile switches that can be used to configure the behavior of coder functions.

Encoding rules configuration

Removing runtime encoding rules availability

- CODER_REMOVE_ASCII - compile coding library without ascii encoding rules available at runtime
- CODER_REMOVE_BER - compile coding library without BER encoding rules available at runtime

- CODER_REMOVE_PER - compile coding library without PER encoding rules available at runtime
- CODER_USE_UER - add user defined encoding and decoding functions to be available at runtime (file “uer.h“ will be used during library compilation)

By default all the library included ascii, ber and per encoding rules to be present in the compiled object library.

Choosing default encoding rules

This compilation switches influence on which encoding rules will be applied when the coder function is called without specifying which type of encoding rules should be applied

- CODER_ER_DEFAULT_PER - use PER as default encoding rules
- CODER_ER_DEFAULT_BER - use BER as default encoding rules
- CODER_ER_DEFAULT_UER - apply USER encoding rules by default

The default setting is CODER_ER_DEFAULT_BER.

- CODER_BER_DEFINITE - use definite length form encoding for BER by default
- CODER_BER_INDEFINITE - use indefinite length form encoding for BER by default

The default setting is CODER_BER_INDEFINITE.

- CODER_PER_NO_ENDPAD - use PER unaligned without end padding as default encoding rules
- CODER_PER_ALIGNED - use PER aligned as default encoding rules
- CODER_PER_UNALIGNED - use PER unaligned as default encoding rules

The default setting is CODER_PER_UNALIGNED.

- CODER_BER_CONSTRUCTED - use constructed form of BER by default

- CODER_BER_PRIMITIVE - use primitive form of BER by default

The default setting is CODER_BER_PRIMITIVE.

- CODER_BER_CONSTRUCTED_LENGTH=<number> - use <number> as length of constructed sequence of bytes for BER

The default setting is CODER_BER_CONSTRUCTED_LENGTH=1000.

- CODER_BER_CANONICAL_ON - restrict BER decode for SET type by canonical order.
- CODER_BER_CANONICAL_OFF - BER decode for SET type as defined in ITU X.690.

The default setting is CODER_BER_CANONICAL_ON.

Real values encoding

For encoding real values are divided into the following number values: (sign * 2^factor * number * base^exponent). It is the user option to choose which factor and which base should be used when encoding.

- CODER_BER_REAL_FACTOR_0 - use real factor 0 for coding
- CODER_BER_REAL_FACTOR_1 - use real factor 1 for coding
- CODER_BER_REAL_FACTOR_2 - use real factor 2 for coding
- CODER_BER_REAL_FACTOR_3 - use real factor 3 for coding
- CODER_REAL_BASE_2 - use real base 2 for encoding
- CODER_REAL_BASE_8 - use real base 8 for encoding
- CODER_REAL_BASE_16 - use real base 16 for encoding

The default setting is CODER_BER_REAL_FACTOR_0 and CODER_REAL_BASE_2.

Error checking configuration

- CODER_CHECK_NONE - remove all error checks from compiled library
- CODER_CHECK_NONE_VALUE - remove error checks of input values to encoding and decoding function from the compiled library

- CODER_CHECK_NONE_BUFFER - remove error checks in buffer management from the compiled library
- CODER_CHECK_NONE_DECODING - remove error checks in decoding functions from the compiled library
- CODER_CHECK_NONE_INNER - remove internal error checks in encoding and decoding functions from the compiled library

> **Note: Error checks in decoding functions**
>
> All decoding functions start with a check of type nodes and input values. These tests are not removed by defining the CODER_CHECK_NONE_DECODING macro. Checks in the buffer management are not affected by the macro either. You can remove these checks by using other macros in this list.

By default all checks are switched ON.

Encoding configuration

You can use the following switches to remove code for ASN.1 type from the compilation:

- CODER_NOUSE_BOOLEAN
- CODER_NOUSE_INTEGER
- CODER_NOUSE_NULL
- CODER_NOUSE_OBJECT_IDENTIFIER
- CODER_NOUSE_REAL
- CODER_NOUSE_BIT_STRING
- CODER_NOUSE_OCTET_STRING
- CODER_NOUSE_CHARACTER_STRING - NumericString, PrintableString, IA5String, VisibleString, UTCTime, GeneralizedTime
- CODER_NOUSE_ENUMERATED
- CODER_NOUSE_ENUMERTAED_ITEMS - ENUMERATED items if they are 0 1 2 3 etc

- CODER_NOUSE_SEQUENCE
- CODER_NOUSE_SET
- CODER_NOUSE_SET_OF
- CODER_NOUSE_SEQUENCE_OF
- CODER_NOUSE_CHOICE
- CODER_NOUSE_OPEN
- CODER_NOUSE_EXT - extension marker in ASN.1 sequence and set

You can use the following switches to redefine C type for ASN.1 type information field:

- CODER_ENUMERATED_TYPE=<type> - enumerated value. Default is “long”
- CODER_TagNumber_TYPE=<type> - ASN.1 tag class number. Default is “unsigned long”
- CODER_NumOf_TYPE=<type> - number of components. Default is “unsigned long”
- CODER_INTEGER_LBOUND_TYPE=<type> - low bound in integer constraint. Default is “long”
- CODER_INTEGER_UBOUND_TYPE=<type> - upper bound in integer constraint. Default is “long”
- CODER_SIZE_LBOUND_TYPE=<type> - low bound in size constraint. Default is “unsigned long”
- CODER_SIZE_UBOUND_TYPE=<type> - upper bound in size constraint. Default is "unsigned long"

These compile switches can be set automatically. `asn1util` generates `asn1_cfg.h` file for the `-c` option where the above compile switches are included after `#define` directives to minimize coder library code size automatically, see also “Configuration file generation” on page 699 in chapter 14, *The ASN.1 Utilities*. To be able to use this `asn1_cfg.h` automatic configuration you should define the following compile switch:

- CODER_AUTOMATIC_CONFIG - enables automatic configuration generated to the `asn1_cfg.h` file. This switch can be set by choosing the corresponding option in the Targeting Expert.

Buffers configuration

- CODER_BMS_SMALLBUF - uses small buffer implementation. The second argument of BufInitBuf interface - type of the buffer (see "tBMSBufType" on page 2777) will not be in use for this case.
- CODER_REMOVE_SMALLBUF - removes small buffer code from the compiled library

The default is including only small buffers to the library. When all buffers are absent, then only a null buffer will be available (a null buffer is a stub buffer without any functionality).

- CODER_USE_USERBUF - includes the user buffer into the library to be available at runtime
- CODER_BMS_USERBUF - uses the user buffer implementation if it is made available by previous switch
- CODER_BMS_TINY - uses a minimum set of bms interfaces. Interfaces to operate with buffer mode, character segment, bytes, bits also copy buffer interface are not available for this switch. It is allowed to use both buffer access types `tBuffer` and `tCoder` in this mode.

Example 469: Encoding in tiny mode

```
tCoder coder;
tBMSUserMemory UserMemory;

BufInitBuf(&coder, bms_SmallBuffer);
ASN1_ENCODE(&coder,
            (tASN1TypeInfo *)&yASN1_Message,
            encValue);
BufCloseBufToMemory(&coder, &UserMemory);
...
BufInitBufWithMemory(&coder, &UserMemory);
ASN1_DECODE(&coder,
            (tASN1TypeInfo *)&yASN1_Message,
            decValue);
BufCloseBuf(&coder);
CUCF_FREE(UserMemory.MemPtr, UserMemory.MemSize, 0);
```

Example 470: `tBuffer` **access type in tiny mode**

```
tBuffer buf = NULL;
BufInitBuf(buf, bms_SmallBuffer);
```

> **Note:**
>
> The buffer value for this mode must be initialized by NULL before being used in buffer management. The behavior will otherwise be unpredictable.

Following buffer interfaces are not available in this mode:

```
BufGetMode, BufInNoMode, BufInReadMode, BufInWriteMode

BufCopyBuf

BufInitWriteMode, BufCloseWriteMode

BufInitReadMode, BufCloseReadMode, BufCloseDeleteReadMode

BufGetByte, BufPeekByte, BufPutByte

BufGetSeg, BufSkipSeg, BufPeekSeg, BufPutSeg

BufPutBit, BufGetBit, BufPutBits, BufGetBits
```

- CODER_SMALLBUF_SIZE=<number> - <number> defines small buffer allocation block

The default setting for the size is CODER_SMALLBUF_SIZE=0x1000.

Memory management configuration

- CODER_MMS_SDT - use SDT alloc/free procedures when working with memory in the coder library functions
- CODER_MMS_USER - apply user defined alloc/free functions when working with memory in the coder library functions (file “mms_user.h“ will be added for library compilation)

By default standard malloc and free functions from <stdlib.h> are used.

Error output configuration

- CODER_EO_SDT - use SDT error output functions (when this switch is turned on, SDT error output switch XECODER can influence error output functionality)
- CODER_EO_USER - apply user defined error output procedures to the library and use them for error output (file “ems_eo_user.h“ will be added for library compilation)
- CODER_EO_DEBUG - use internal library functions for printing error messages
- CODER_EO_NONE - do not output error messages text

The default setting is CODER_EO_SDT.

- CODER_TI_NAMES - enable name information in the Type Info.

Error handling configuration

- CODER_PATH_DEEP=<number> - nesting coefficient. It depends on the ASN.1 specification. The nesting coefficient is the maximum number of nested SEQUENCE, SET, CHOICE, SEQUENCE OF, SET OF or open types. By default `CODER_PATH_DEEP` is defined to 16. If the nesting coefficient in the ASN.1 specification is greater then 16 then you should define the `CODER_PATH_DEEP` compile switch with this value yourself, otherwise “segmentation fault” error will occur. The nesting coefficient for the following example is 4:

```
MySeq ::= SEQUENCE { -- NC=4
  a MyChoice,
  b NULL
}
MySet ::= SET { -- NC=1
  a INTEGER,
  b BOOLEAN
}
MyChoice ::= CHOICE { -- NC=3
  a MySeqOf,
  b MySetOf
}
MySeqOf ::= SEQUENCE OF MySet -- NC=2
MySetOf ::= SET OF BOOLEAN -- NC=1
```

- CODER_REMOVE_PATH - do not use detailed error messages (see “Detailed error messages” on page 2797) and user error handling (see “User defined Error Handling” on page 2806)

User data configuration

- CODER_USER_DATA - include `tUserData` defined in the `user_data.h` into the coder buffer, see “User Data” on page 2810

Value management configuration

- CODER_VMS_SDT - use SDT value interface when accessing external values from the encoding and decoding functions (this switch comes together with SDT switch XUSE_GENERIC_FUNC, which chooses generic value representation out of two available value representations for SDT/C values, by default it is turned off and old (backwards compatible) value representation is used)
- CODER_VMS_TTCN - use TTCN value interface when accessing external values from the encoding and decoding functions
- CODER_VMS_USER - choose user value interface for accessing external values from the encoding and decoding functions (file “vms_vr_user.h“ must be edited to configure user value access, it is included for library compilation)

By default CODER_VMS_SDT is chosen.

Printing configuration

- CODER_VMS_PRINT - enable debug printing opportunities (see “Printing Opportunities” on page 2811)

By default the coder library is compiled without debug printing functionality.

Generating Environment Files with Coding

You can call the encoding and decoding functions from the environment functions if you do not want to manipulate the encoded bit pattern explicitly in your SDL system.

The SDL Suite tools generate templates for the environment files that contains all necessary calls to buffer management functions, encoding functions and decoding functions. You can, of course, adapt this template to your needs or even write all the calls yourself. See chapter 58, *Building an Application* or chapter 66, *The Cmicro SDL to C Compiler*.

The template solution in the environment files has the following properties:

- One global buffer reference created at start-up of system. This buffer reference is used in all calls for all signals and for both encoding and decoding.
- All signals out from the system are encoded into bit-patterns. All parameters of the signals are encoded.
- All signals into the system decodes bit-patterns.

Compiling and Linking

The instructions you must perform in order to compile and link are listed in chapter 8, *Tutorial: Using ASN.1 Data Types*, *in the SDL Suite Getting Started*.