Europay

Open Terminal Architecture (OTA) Specification

Volume 3: C Language Programming Interface

Europay International S. A
198A Chaussée de Tervuren
1410 Waterloo, Belgium

Version 3.2 – 09 December 1998

About This Book

Purpose

The Europay Open Terminal Architecture (OTA) consists of technology designed to facilitate implementation of Integrated Circuit Cards (ICCs) and associated payment terminals.

The purpose of this document is to provide a specification for a C language interface to a standard kernel to be provided in all OTA payment terminals.

Scope

OTA defines a standard software kernel whose functions and programming interface are common across all terminal types. This kernel is based on a standard "Virtual Machine," which is implemented on each CPU type and which provides drivers for the terminal's I/O and all low-level CPU-specific logical and arithmetic functions. High-level libraries, terminal programs and payment applications may be developed using these standard kernel functions.

The specification of the OTA Virtual Machine is given in Volume 1 of this series. This volume contains the C language bindings for kernel functions, as well as specifications for C language compliant compilers.

Audience

This document is addressed to C developers and contains the specific information needed to write programs for the OTA environment in the C language. The reader should be familiar with the concepts and notation described in OTA Volume 1 and the use of the OTA C token compiler.

The reader is referred to the OTA Volume 1 for the complete description of the OTA virtual machine and to OTA Volume 4 for addressing mixed-language issues and inter-module communication.

Document Change Control

The information in this manual supersedes and replaces all previous drafts issued.

Additional Copies

Requests for copies of Europay publications should be made to:

Europay Documentation Centre
198A Chaussée de Tervuren
1410 Waterloo
Belgium

How This Book is Organised

This document is divided into the following chapters:

Chapter 1 *OTA Virtual Machine interface interface* describes the interface to "Virtual Machine" architecture upon which OTA is based.

Chapter 2 *System services* describes the various services provided by an OTA kernel to client programs, their C implementation and the associated C data structures.

Chapter 3 *C Library Reference* describes the detail of each function implementing the services provided by an OTA kernel to client programs.

Aids in Using This Book

This manual contains the following aids for using the information it presents:

- A list of all of the tables present in this book, found on page ix.
- An alphabetical list of OTA functions, with page numbers, in Appendix A.
- A summary of exception codes and I/O result codes in Appendix A.
- An overview of the device control functions in Appendix B.
- An overview of the operating system functions in Appendix C.
- An index of topics covered in this book, found at the end of the document.

Related Publications

The following Europay publications contain material directly related to the content of this book.

- *Integrated Circuit Card Specification for Payment Systems*, Version 3.0, June 30, 1996
- *Integrated Circuit Card Terminal Specification for Payment Systems*, Version 3.0, June 30, 1996
- *Integrated Circuit Card Application Specification for Payment Systems*, Version 3.0, June 30, 1996
- *Open Terminal Architecture (OTA) System Overview*, Version 2.1, May 19, 1997
- *OTA C Token Compiler User's guide*, Version 1.2, July, 1997

- *OTA Specification Volume 1: Virtual Machine Specification*, Version 3.2, 14 May, 1998
- *OTA Specification Volume 2: Forth Programming Language Interface*, Version 2.0, May, 1997
- *OTA Specification Volume 4: Mixed language Programming*, Version 1.1, July 1997

Reference Materials

The following reference materials may be of use to the reader of this manual:

B.W. Kernighan and D.M. Ritchie	The C Programming Language, 2nd edition
ANSI X9.30-2:1993	Public key cryptography using irreversible algorithms for the financial services industry — Part 2: The Secure Hash Algorithm (SHA)
ANSI X9.31-1 (Draft)	Public key cryptography using irreversible algorithms for the financial services industry — Part 1: The RSA Signature Algorithm
FIPS PUB 180-1:1994	Secure Hash Standard
ISO 639:1988	Codes for the representation of names and languages
ISO 3166:1993	Codes for the representation of names of countries.
ISO 4217:1990	Codes for the representation of currencies and funds.
ISO/IEC 8825:1990	Information technology — Open systems interconnection — Specification of basic encoding rules for abstract syntax notation one (ASN.1).

Table of Contents

List of Tables

1 OTA Virtual Machine interface

1.1 Introduction

This chapter discusses some implementation features of the C language on OTA in order to help the C developer to debug and optimise the code on the target environment. It is not necessary for C programmers to know the implementation details of the OTA C token compiler, however it may help solving performance issues.

1.2 Entry point

In the OTA environment C programs are compiled into modules. A module can be an executable module or a library module. A library module contains a number of functions that are exported and used by other modules. An executable module contains an *entry point*. This is the function that is called if the module is executed. It is the main program of the module. The default entry point of an OTA C module is named `app` and not `main` as in standard ANSI C programs.

1.3 C Data Types and use of the Frame Stack

1.3.1 Space allocation on the Frame Stack

C local variables are implemented by the OTA C token compiler using the Frame mechanism (see OTA Volume 1, Section 3.2.5).

OTA standard data types are described in OTA Volume 1, table 1. The storage unit in OTA is the stack cell, which size is 4 bytes.

The space used on the Frame stack for the C basic data types is summarised below:

Type	Size
characters and character arrays	1/4 cell per character
numbers: short int, int, long int and arrays of numbers	1 cell per number
pointers	1 cell per pointer

Table 1: Space used on the Frame stack for the C basic data types

Example

For the simple C program below using a local array of 4 characters, one cell is allocated on the frame stack for the four characters.

```
void app(void)
{
   char str[4];
   str[0]='a';
   str[1]='b';
   str[2]='c';
   str[3]='d';
}
```

Description of the optimised code generated by the OTA C token compiler

`smkframe 0 1`	• create a frame: 0 cell for the procedure parameters, 1 cell for the four temporary character variables
`slit 97`	• put the ASCII value of the 'a' character on the data stack
`byte sfrstore -4`	• fetch the character on the data stack and store it into the byte at the offset -4 in the active frame
`slit 98`	• put the ASCII value of the 'b' character on the data stack
`byte sfrstore -3`	• fetch the character on the data stack and store it into the byte at the offset -3 in the active frame
`slit 99`	• put the ASCII value of the 'c' character on the data stack
`byte sfrstore -2`	• fetch the character on the data stack and store it into the byte at the offset -2 in the active frame
`slit 100`	• put the ASCII value of the 'd' character on the data stack
`byte sfrstore -1`	• fetch the character on the data stack and store it into the byte at the offset -1 in the active frame
`relframe`	• restore the frame pointer and release the current frame
`return`	• return from the procedure

1.3.2 String representation

C proposes a very simple convention to represent and manipulate ASCII strings: the address of a character cell supposed to be the first character of an array. The end of the string is marked by a binary `NULL`. Null-terminated strings can be used in the OTA virtual machine.

However, most of the data manipulated by the applications running on OTA are binary strings, where `NULL` values are a significant part of the data, and not string terminators. For manipulating such strings, counted strings should be used.

The OTA standard representation for strings uses counted strings (see OTA Volume 1: *c-addr len*). For this reason, most of the functions of the OTA C library work on the `BUF_Object` datatype (see Section 2.5) to represent counted strings.

Example

For the C program below using an initialised array of 4 characters, one cell is allocated on the frame stack for the address of the *str* character pointer.

```
void app(void)
{
   char *str = {"ABCD"};
}
```

Description of the optimised code generated by the OTA C token compiler

`smkframe 0 1`	• create a frame: 0 cell for the procedure parameters, 1 cell for the character pointer
`slitd0 0`	• push the address +offset 0 of the uninitialised data section on the data stack
`tfrstore 1`	• fetch the cell on the data stack and store it into the cell at the offset -4 in the active frame
`relframe`	• restore the frame pointer and release the current frame
`return`	• return from the procedure

1.3.3 Alignment

The OTA Virtual Machine requires that aligned addresses be exact multiples of 4. The C compiler uses aligned addresses to store integers. Therefore it is possible that the size of a `struct` is greater than the sum of the sizes of the fields of the `struct`. One should always use the `sizeof` operator to define the size of a `struct`.

Example

The following structure will have a size of 8 bytes, because the compiler aligns the integer field to a second cell.

```
typedef struct {
   char str[3];
   int i;
} my_struct;
```

1.4 Data Stack and Function parameters

1.4.1 Return Parameters

The OTA C token compiler uses the data stack to pass return values to the calling function. In the following example, the constant 0 is returned by the function `funct1` to the main application.

```
int funct1(void)
{
   return 0;
}

void app(void)
{
   funct1();
}
```

The OTA C token compiler generates the following optimised code:

`smkframe 0 0`	• create a frame for *funct1*: 0 cell for the procedure parameters, 0 cell for the local variables
`lit0`	• put the constant 0 on the data stack
`relframe`	• release the frame
`return`	• return from the procedure
`smkframe 0 0`	• create a frame for the *app* procedure: 0 cell for the procedure parameters, 0 cell for the local variables
`scall 0`	• call *funct1* at offset 0 from the beginning of the module
`drop`	• remove the return value from the data stack
`relframe`	• release the frame
`return`	• return from the procedure

1.4.2 Calling Parameters

The OTA C token compiler uses the data stack to pass parameters to the called function.

Example

The constants 1, 2, 3 and 4 are passed to the function `plusminus` by the main application.

```
int plusminus(int i,int j,int k,int l)
{
   return(i+j+k-l);
}

void app(void)
{
   plusminus(1,2,3,4);
}
```

The OTA C token compiler generates the following optimised code:

`smkframe 4 0`	• create a frame for *plusminus* : 4 cells for the procedure parameters, integer i,j,k and l, 0 cell for the local variables
`pfrfetch2`	• put the contents of *i* on the data stack (frame cell FP+8)
`pfrfetch3`	• put the contents of *j* on the data stack (frame cell FP+12)
`add`	• add
`pfrfetch4`	• put the contents of *k* on the data stack (frame cell FP+16)
`add`	• add
`pfrfetch5`	• put the contents of *l* on the data stack (frame cell FP+20)
`sub`	• subtract
`relframe`	• release the frame
`return`	• return from the procedure

`smkframe 0 0`	• create a frame for the *app* procedure: 0 cell for the procedure parameters, 0 cell for the local variables
`lit4`	• put value 4 for parameter *l* on the data stack
`lit3`	• put value 3 for parameter *k* on the data stack
`lit2`	• put value 2 for parameter *j* on the data stack
`lit1`	• put value 1 for parameter *i* on the data stack
`scall 0`	• call *plusminus* at offset 0 from the beginning of the module
`drop`	• remove the return value from the data stack
`relframe`	• release the frame
`return`	• return from the procedure

Stack effect

The call `plusminus(1,2,3,4)` puts the values on the data stack in the reversed order (the first declared parameter is the last parameter being put on the stack).

1	Top of Stack
2	
3	
4	

The OTA equivalent notation for

```
sum = plusminus(i,j,k,l);
```

is

```
plusminus (l k j i -- sum )
```

1.5 Dynamic Memory Allocation

The OTA C token library provides access to the Virtual Machine extensible memory through functions similar to the `malloc` and `free` functions of the ANSI-C. These OTA C functions are: `MEM_extend`, `MEM_extend_bytes` and `MEM_release`.

There is a major difference between the ANSI and the OTA functions.

In ANSI-C, the memory blocks reserved by `malloc` and `calloc` are allocated independently and may be released in any order.

In OTA, `MEM_extend` adds a contiguous block to the block of extensible memory currently allocated and returns the address of the new block, while `MEM_release` releases the corresponding block, and all blocks allocated after it.

Example

```
#include <mem.h>
#define BLOCK_LENGTH 16
```

```
void app(void)
{
   char * str = "0123456789abcdef";
   void * block1, * block2, * block3;

   // allocate 3 blocks of extensible memory
   block1 = MEM_extend(BLOCK_LENGTH);
   block2 = MEM_extend(BLOCK_LENGTH);
   block3 = MEM_extend(BLOCK_LENGTH);

   // use block 1
   MEM_move(block1, str, BLOCK_LENGTH);

   // release block 2
   MEM_release(block2);

   // try to use block 3 :
   // the next instruction generates an 'invalid memory addres'
   // exception throw. The block 3 has been released together with block 2
   MEM_move(block3, str, BLOCK_LENGTH);
}
```

1.6 ANSI C standard libraries

The list of the functions of the ANSI-C standard library supported and not supported by the OTA C token library and the functions of the OTA C token library similar to ANSI functions is given in this section.

ANSI Library	Functionality	Support in OTA token library
<stdio.h>	file functions	not supported but see database handling
	formatting functions	only sprintf is supported
	character I/O	not supported but see <dev.h>
	stream I/O	not supported
	error handling	see <xcp.h> (catch & throw)
<mem.h>	memchr	supported
	memcpy	supported (synonym of *MEM_move*)
	memmove	supported (synonym of *MEM_move*)
	memset	supported (synonym of *STR_fill*)
<string.h>	strlen	supported (synonym of *STR_length*)
	strcpy	supported
	strncpy	supported
	strcat	supported
	strncat	supported
	strchr	supported
	strcmp	supported
	strncmp	supported

Table 2: Ansi C standard libraries

ANSI Library	Functionality	Support in OTA token library
<stdlib.h>	number conversion	see <str.h>
	memory allocation	see <mem.h>
	other	in general not supported: except sleep see *TMR_delay*
<assert.h>	debug	see <dbg.h>
<time.h>	time and date	see <tmr.h>
<setjmp.h>	unconditional branch	see <xcp.h>
<ctype.h>	character type	not supported
<math.h>	math functions	not supported, some other specialised OTA mathematical functions are implemented in <sec.h>.
<float.h>	floating point	not supported
<signal.h>	signal handling	not supported

Table 2: Ansi C standard libraries (continued)

2 System services

2.1 Naming Conventions

Functions which are specific to the OTA C token library are prefixed by three letters. The three letters are either an abbreviation or an acronym for the purpose of the function. For example, the MAG abbreviation is used by magnetic stripe functions, and the HCF acronym is used by hot card file functions.

Every OTA C token library function has a prototype in a header file. The header file to include at the top of the source file is the abbreviation or acronym with the standard ".h" suffix. For example, to access the hot card file functions whose prefix is "HCF", you would use

```
#include <hcf.h>
```

in the C source file.

Macro definitions should always be created in upper case.

2.2 Database services

The OTA database services implement record management facilities described in OTA Volume 1. They are discussed in this section in the scope of the C language implementation.

Database Parameter Block (DPB)

The structure of a Database Parameter Block (DPB) described here is defined in the `<dbs.h>` file. This structure allows the user to create and use databases in a C module and to share them with other modules.

```
typedef struct DBS_pb {
   struct DBS_pb *link;
   void *data;
   unsigned int type;
   unsigned size;
   unsigned available;
   unsigned key_offset;
   unsigned key_size;
} *DBS_Handle;
```

Database Creation

Databases are defined at compile-time and instantiated by the loading process for the module in which its DPB is defined. At load time, the OTA Virtual Machine (VM) reads the information containd in all DPBs starting from the `DBS_root` `DBS_pb` type structure, which is the

reserved name of the C compiler corresponding to the MDF-DBROOT (see OTA Volume1, Section 6, Module delivery format). Depending on the type of the database, the VM links the DPB to the non-volatile data, to the compiled data or assigns volatile data space to the database.

Local Database Access

The OTA concept of current database and current record are hidden in the C interface. All C database access functions specify the address of the Database Parameter Block and eventually the record number to be accessed (first record is record #0).

Accessing the database from other modules

Full access can be given to a database by exporting the database handle (see OTA C Token Compiler User's manual, Section 7.4, Exporting variables between modules).

Another way of giving a more restricted access to a database is to make the database temporarily current; the functions of other modules will be given access to the current database only.

The most restrictive method is to provide access functions within the module which owns the database and to export the functions.

Record Access

The internal structure of the fields in the record is independant from the database definition. The functions accessing the records by the record number use a pointer to a `void` to pass the address of the record. This pointer may be cast to a structure to provide access to the internal structure of the record. For ordered databases, the key is passed in a `BUF_object`.

Database types

The following constants are defined in `<dbs.h>` and can be used to initialise the `DBS_pb.type` element:

```
DBS_VOLATILE vs DBS_NON_VOLATILE,
DBS_INITIALIZED,
DBS_NOT_ORDERED vs DBS_ORDERED,
DBS_RW vs DBS_RO
```

DPB initialisation rules

Compiled database

`DBS_pb.link`	address of DPB of next database, `NULL` if last database
`DBS_pb.data`	address of first character of initialised data array
`DBS_pb.type`	`DBS_INITIALIZED` [| <other compatible attribute>]
`DBS_pb.size`	number of characters of one record
`DBS_pb.available`	constant : total number of records in the initialised database

Ordered database

`DBS_pb.link`	address of DPB of next database, `NULL` if last database.

`DBS_pb.data`	address of first character of initialised data array, `NULL` if database not initialised
`DBS_pb.type`	`DBS_ORDERED` [| <other compatible attribute>]
`DBS_pb.size`	number of characters of one record
`DBS_pb.available`	`NULL` if the database is not initialised , otherwise total number of records in the initialised database
`DBS_pb.key_offset`	byte offset within the record at which the key begins
`DBS_pb.key_size`	size in bytes of the key within the database record

volatile or non-volatile database

`DBS_pb.link`	address of DPB of next database, `NULL` if last database.
`DBS_pb.data`	`NULL`
`DBS_pb.type`	`DBS_VOLATILE` vs `DBS_NON_VOLATILE` [|<other compatible attribute>]
`DBS_pb.size`	number of characters of one record
`DBS_pb.available`	`NULL`

Example

```
// define a compiled database with 4 records, read and display the four
// records, display the total number of records

#include <dbg.h>
#include <dbs.h>
#include <mem.h>

#define LEGEND_SIZE 16

typedef struct {
   char field_one[3];
   char legend[LEGEND_SIZE];
   char recnr;
} legend_rec;

legend_rec recs[] = {
   {"123","legend first rec",'1'},
   {"456","legend secnd rec",'2'},
   {"789","legend third rec",'3'},
   {"ABC","legend forth rec",'4'}
};

DBS_define( DBS_root,
   0,                      // link to next DPB
   recs[0].field_one,      // address of data
   DBS_INITIALIZED,        // database type
   sizeof (legend_rec),    // size of each record
   4,                      // compiled, 4 records
   0,                      // key
   0                       // key length
);
```

```
// display the contents of the DBS handle
void Type_DPB(DBS_Handle DBS_ptr)
{
   DBG_CR; DBG_STR(" ptr: ");         DBG_INT((int)DBS_ptr);
   DBG_STR(" link: ");                DBG_INT((int)DBS_ptr->link);
   DBG_STR(" size of record: ");      DBG_INT(DBS_ptr->size);
   DBG_STR(" type of db ");           DBG_INT(DBS_ptr->type);
   DBG_STR(" next record available: "); DBG_INT(DBS_ptr->available);
   DBG_STR(" key_offset: ");          DBG_INT(DBS_ptr->key_offset);
   DBG_STR(" size: ");                DBG_INT(DBS_ptr->key_size);
   DBG_STR(" contents: ");            DBG_STR(DBS_ptr->data);
}

void demo_db_read(void)
{
   char string[22];
   Type_DPB(&DBS_root);
   DBS_read(&DBS_root,0,string);
   DBG_CR; DBG_STR("record0: "); DBG_CSTR_ASCII(string,16);
   DBS_read(&DBS_root,1,string);
   DBG_CR; DBG_STR("record1: "); DBG_CSTR_ASCII(string,16);
   DBS_read(&DBS_root,2,string);
   DBG_CR; DBG_STR("record2: "); DBG_CSTR_ASCII(string,16);
   DBS_read(&DBS_root,3,string);
   DBG_CR; DBG_STR("record3: "); DBG_CSTR_ASCII(string,16);
}

void app(void)
{
   DEV_open(DEV_DEBUG);
   demo_db_read();
   DBG_CR; DBG_STR("total records: ");
   DBG_INT(DBS_avail(&DBS_root));
   DEV_close(DEV_DEBUG);
}
```

2.3 TLV services

The OTA C token library provides functions which access and manipulate the values contained in TLVs (abbreviation for Tag-Length-Value).

Each TLV is named using an integer called the *tag*. Each TLV has associated with it a *length*, which is the length of the data held in the TLV. The *value* is the contents of the data of the TLV.

TLV types

The C representation of TLV types uses an enum-type variable defined in `<tlv.h>`.

```
enum TLV_type {
   TLV_N,    // BCD nibbles padded with leading zeroes
   TLV_B,    // byte sequence
   TLV_BN,   // binary number, msb first byte
   TLV_CN,   // compressed numeric, bcd nibbles + trailing ff's
```

```
    TLV_AN,   // alphanumeric characters + trailing zeroes
    TLV_ANS,  // full ASCII characters with trailing zeroes
    TLV_VAR   // variable format
};
```

Representation of the TLV value

TLVs fall into two flavours, depending on their type:

- `TLV_N` and `TLV_BN` which contain numbers which can be represented in an unsigned 32-bit word,
- all the other types are described by an address and a length.

TLV Database structure

All the TLVs defined in a module are stored in volatile memory; OTA uses a binary tree structure to allow an efficient retrieval of a tag within the structure.

The structure of a node of the tree is defined in `<tlv.h>` :

```
typedef struct TLV_node {
   unsigned int link;
   unsigned int def;
   unsigned int data;
} tlv_n, *TLV_t;
```

used as follows :

0 8 16 24 32

link left : 16-bit signed offset	link right	
tag	type	reserved
pointer to data		

TLV definitions and data can be physically distributed across several modules**.** All the TLV tree structures are linked to each other as a unique TLV database accessible from all the modules. The link between all the tree structures is set by the OTA kernel at load time.

TLV Definitions

The TLVs of a module are defined at compile time and are linked to the TLV database at load time. To facilitate the balancing of the tree and the calculation of the offsets, a pre-processing tool generates assembler code from standard TLV definitions. The code generated can be compiled and linked with the other object modules to create the mdf file. The ‘TLV pre-processor’ creates also the definitions needed to access the TLVs from Forth code. (see OTA Volume 4)

Example

The following code can be stored in a `<.tlv>` file and used to generate the TLV definition code.

```
\ ------------------------------------------------------------------------
\     TLV definitions;  file name : EMVDEF.TLV
\ ------------------------------------------------------------------------
\ Format  Tag       Name             Description
\ ------  ---       ----             -----------
  TLV-B   $4F  TLV: TLV-ICC-AID      \ Application Identifier (AID) (ICC)
  TLV-AN  $50  TLV: TLV-APP-LABEL    \ Application Label
  TLV-B   $52  TLV: TLV-COMMMAND     \ Command to Perform
```

Using TLVs which contain numbers

Let us first consider working with TLVs which hold 32-bit numbers. Such a TLV is the TLV which contains the terminal floor limit, which has a tag 0x9F1B. The two functions used to read and write this TLV are `TLV_fetch` and `TLV_store`:

```
unsigned TLV_fetch(int tag);
void TLV_store(int tag, unsigned val);
```

These functions take a tag and return the data held in that TLV. So, to increment the terminal floor limit by two, we can use the following:

```
#include <tlv.h>
#define TLV_TERMINAL_FLOOR_LIMIT 0x9F1B
void app(void)
{
   TLV_store(TLV_TERMINAL_FLOOR_LIMIT,
      TLV_fetch(TLV_TERMINAL_FLOOR_LIMIT)+2);
}
```

Using TLVs which contain byte streams

Functions handling TLVs representing strings or numeric values larger than a 32-bit word use two global variables defined in `<tlv.h>`.

```
#define TLV_MAX_LENGTH 257
int TLV_length;
char TLV_value[TLV_MAX_LENGTH];
```

The `TLV_length` and `TLV_value` pair of global variables are the single interface to access the TLV database for the corresponding TLV types. They should be used as a temporary buffer only and their contents should be saved in another variable after reading.

```
#include <tlv.h>
#include <mem.h>
#define TLV_APP_LABEL (int)(0x050)
#define TLV_CARDHOLDER_NAME (int)(0x5F20)

char str[257];

// write the string to be stored in the TLV in the TLV_value buffer
MEM_move(TLV_value[0],"APPL1",5);
// Set the length of the string
TLV_length = 5;
// write into the TLV database
TLV_put(TLV_APP_LABEL);
// read another TLV from the database
TLV_get(TLV_CARDHOLDER_NAME);
// save the contents into str
MEM_move(str, TLV_value, TLV_length);
```

2.4 Device I/O Services

Device I/O services are divided in three libraries:

- DEV - Generic Device I/O including keyboards, displays, printers and modems;
- ICC - IC Card reader I/O;
- MAG - Magnetic Stripe Card reader I/O.

The IO device functions always take the device number as an input parameter. See Appendix B, Table 26 for the device code assignments. The device number is represented by the `DEV_Handle` enumerator type and defined in `<dev.h>`.

All device access functions return an I/O return code (ior) but may also throw. See Appendix B for the different I/O return codes. The I/O return codes are implemented as integers and their values are defined in `<xcp.h>`. A device access function throws on a global error, but returns an ior for situatioins that are correctable at or near the device level. Note that for all the device functions the generic throw code -21 (`XCP_UNSUPPORTED_OPERATION`) is returned as an ior and not thrown as an exception.

2.5 Buffer Objects

There are a number of functions in the OTA C token library which operate on buffer objects. Buffer objects are data structures used for holding data of variable length within a fixed-size block. A buffer object is defined by three fields:

- *len*, the current size of the data held in the buffer, also known as the buffer length;
- *size*, the maximum size of data which can be held in the buffer;
- *data*, a pointer to the data held in the buffer.

Structure

The C structure implementing buffer objects is described below and is defined in `<buf.h>`.

```
typedef struct {
   int len;
   int size;
   char *data;
}  BUF_Object, *BUF_Handle;
```

The specification of the functions working with buffer objects can be found in Section 3.1.

2.6 Exception handling

Errors which occur during execution of OTA C token programs usually throw exceptions. Each exception has a unique number associated with it, and a comprehensive list of these can be found in the header file `<xcp.h>`.

The list of throw codes is divided in two categories : OTA specific throw codes (Table 23)

and generic throw codes (Table 24). Every library function may throw a generic throw code under the given condition, and they are not listed individually under those functions. The OTA specific throw codes are documented for every function individually.

2.7 Debug functions

Some functions and macros are added in order to help debugging of programs. They allow to write the contents of various types of variables or control characters as integer, hexadecimal number, null-terminated string, `BUF_Object` buffers on the standard DEBUG device.

The `DBG_ASSERT` and `ASSERT` macros perform a throw if a condition is not satisfied.

Macro names are written in upper case (`DBG_INT`, `DBG_STR`,...). Macro functions are added to the object code when the DEBUG pre-processor constant is defined.

The definitions of all these function can be found in Section 3.2

3 C Library Reference

3.1 Buffer Handle functions

	C function	Main Token invoked
3.1.1	BUF_alloc	CEXTEND
3.1.2	BUF_append	MOVE
3.1.3	BUF_append_char	CSTORE
3.1.4	BUF_append_counted_str	MOVE
3.1.5	BUF_append_num	MOVE
3.1.6	BUF_append_str	MOVE
3.1.7	BUF_release	RELEASE
3.1.8	BUF_set_counted_str	MOVE
3.1.9	BUF_set_str	MOVE

Table 3: BUFFER Handle functions

3.1.1 BUF_alloc

If the data field of *buf* is zero, allocate *buf->size* bytes and assign the address of the allocated memory to *buf->data*. *buf->len* is set to zero. If the data field is not zero, no allocation is done and *buf* is left unchanged.

void BUF_alloc (BUF_Handle *buf***);**

Return Value

none

Parameters

input	*buf*	pointer to a BUF_Object

Exceptions

`XCP_OUT_OF_MEMORY`	there is insufficient memory available

Remark

`BUF_alloc` adds a contiguous block to the block of extensible memory currently allocated, while `BUF_release` releases the corresponding block, and all blocks allocated after it (see Section 1.5).

Example

```
#include <buf.h>
#include <dbg.h>
void app(void)
{
   BUF_Object buf_dyn = {0,20,0};
   DEV_open(DEV_DEBUG);
   BUF_alloc(&buf_dyn);
   BUF_set_str(&buf_dyn,"HELLO");
   DBG_CR;
   DBG_STR("buf_dyn = ");
   DBG_DUMP(&buf_dyn);
}
```

3.1.2 BUF_append

Copy every byte of the data field of the append buffer at the end of the data field of the destination buffer. Raise an XCP_OUT_OF_MEMORY exception if the size of *buf_dest* is smaller than the original length of destination buffer + the length of the append buffer.

void BUF_append (BUF_Handle *buf_dest***, BUF_Handle** *buf_append***)**

Return Value

none

Parameters

input	*buf_append*	contains the string to be appended
output	*buf_dest*	destination buffer

Exceptions

XCP_OUT_OF_MEMORY	insufficient space left in *buf_dest*

Example

```
#include <buf.h>
#include <dbg.h>
void app(void)
{
   char str_dest[20];
   char str_dest2[20];
   BUF_Object buf = {0,20,0};
   BUF_Object buf2= {0,20,0};
   buf.data = str_dest ;
   buf2.data = str_dest2;
   DEV_open(DEV_DEBUG);
   BUF_set_str(&buf,"Hello");
   BUF_set_str(&buf2," World");
   BUF_append(&buf,&buf2);
   DBG_CR;
   DBG_STR("buf after append function = ");
   DBG_DUMP(&buf);
   DEV_close(DEV_DEBUG);
}
```

3.1.3 BUF_append_char

Append the character *c* at the end of the buffer *buf*.

void BUF_append_char (BUF_Handle *buf***, char** *c***)**

Return Value

none

Parameters

input	*buf*	the allocated buffer
	c	the character to append
output	*buf*	increased buffer

Exceptions

XCP_OUT_OF_MEMORY	size of the buffer smaller than the original length of the input buffer + 1

Example

```
#include <buf.h>
#include <dbg.h>
void app(void)
{
   char str[20];
   BUF_Object buf = {0,20,0};
   buf.data = str;
   DEV_open(DEV_DEBUG);
   BUF_set_str(&buf,"Hell");
   BUF_append_char(&buf,'o');
   BUF_append_char(&buf,0);
   DBG_CR;
   DBG_STR("buf after append function = ");
   DBG_STR(buf.data);
   DEV_close(DEV_DEBUG);
}
```

3.1.4 BUF_append_counted_str

Append the counted string pointed by the address *str* and with length *len* to the buffer object pointed by *buf_dest*. Raise an XCP_OUT_OF_MEMORY exception if the size of the destination buffer is smaller than the original length of the destination buffer + the length of the string.

void BUF_append_counted_str (BUF_Handle *buf_dest***, char** **str,* **int** *len***);**

Return Value

none

Parameters

input	*str*	address of the string to be appended

	len	length of the string to be appended
output	*buf_dest*	output buffer object

Exceptions

XCP_OUT_OF_MEMORY	insufficient space left in *buf_dest*

Example

```
#include <buf.h>
#include <mem.h>
#include <dbg.h>
void app(void)
{
   char str_dest[20];
   char str_src[10] ;
   BUF_Object buf = {0,20,0};
   buf.data = str_dest;
   DEV_open(DEV_DEBUG);
   BUF_set_str(&buf,"Hello");
   MEM_move(str_src," World",6);
   BUF_append_counted_str(&buf,str_src,6);
   DBG_CR;
   DBG_STR("buf = ");
   DBG_DUMP(&buf);
   DEV_close(DEV_DEBUG);
}
```

3.1.5 BUF_append_num

Convert the number *n* to an ASCII string and append the string to the buffer object pointed by *buf_dest* buffer. Raise an XCP_OUT_OF_MEMORY exception if the size of *buf_dest* is smaller than the original length of destination buffer + the length of the string.

void BUF_append_num (BUF_Handle *buf_dest***, int** *n***);**

Return Value

none

Parameters

input	*n*	value to be appended
output	*buf*	output buffer object

Exceptions

XCP_OUT_OF_MEMORY	insufficient space left in *buf_des*t

Example

```
#include <buf.h>
#include <dbg.h>
void app(void)
{
   char str_dest[20];
   BUF_Object buf = {0,20,0};
   buf.data = str_dest ;
```

```
    DEV_open(DEV_DEBUG);
    BUF_set_str(&buf,"Hello");
    BUF_append_num(&buf,1);
    DBG_STR("buf = ");
    DBG_DUMP(&buf);
    DEV_close(DEV_DEBUG);
}
```

3.1.6 BUF_append_str

Append the null-terminated string pointed by *str* to the buffer object pointed by *buf*. Raise an XCP_OUT_OF_MEMORY exception if the size of the destination buffer is smaller than the original length of the destination buffer + the length of the string.

void BUF_append_str (BUF_Handle *buf***, char** **str**);**

Return Value

none

Parameters

input	*str*	null terminated string to be appended
output	*buf*	output buffer object

Exceptions

XCP_OUT_OF_MEMORY	insufficient space left in *buf*

Example

```
#include <buf.h>
#include <dbg.h>
void app(void)
{
   char str_dest[20];
   BUF_Object buf = {0,20,0};
   buf.data = str_dest ;
   DEV_open(DEV_DEBUG);
   BUF_set_str(&buf,"Hello ");
   BUF_append_str(&buf,"World");
   DBG_CR;
   DBG_STR("buf after append = ");
   DBG_DUMP(&buf);
   DEV_close(DEV_DEBUG);
}
```

3.1.7 BUF_release

If *buf->data* != 0, release the memory in the data field of the buffer. *buf->data* and *buf->len* are set to zero. If *buf->data* = 0, no release is done and *buf* is left unchanged.

void BUF_release (BUF_Handle *buf***);**

Return Value

none

Parameters

input	*buf*	pointer to a `BUF_Object`

Remark

`BUF_alloc` adds a contiguous block to the block of extensible memory currently allocated, while `BUF_release` releases the corresponding block, and all blocks allocated after it (See “Dynamic Memory Allocation” (page –15)).

Example

```
#include <buf.h>
#include <dbg.h>
void app(void)
{
   BUF_Object buf_dyn = {0,20,0} ;
   DEV_open(DEV_DEBUG);
   BUF_alloc(&buf_dyn);
   BUF_set_str(&buf_dyn,"HELLO");
   DBG_CR; DBG_STR("buf before release = ");
   DBG_DUMP(&buf_dyn);
   BUF_release(&buf_dyn);
   DBG_CR; DBG_STR("buf = ");
   // The next dump won't display anything, buf_dyn.data pointer is now
   // null
   DBG_DUMP(&buf_dyn);
   DEV_close(DEV_DEBUG);
}
```

3.1.8 BUF_set_counted_str

Copy the counted character string in the data field of the buffer. Raise an `XCP_OUT_OF_MEMORY` exception if the length of the string is greater than the size of the destination buffer.

void BUF_set_counted_str (BUF_Handle *buf*, **char ***_str_, **int** *len* **);**

Return Value

none

Parameters

output	*buf*	output buffer object
input	*str*	string to be appended
	len	string length

Exceptions

`XCP_OUT_OF_MEMORY`	*len* greater than *buf->size*

Example

```
#include <buf.h>
#include <mem.h>
#include <dbg.h>
```

```
void app(void)
{
   char str_src[10];
   char str_dest[20];
   BUF_Object buf = {0,20,0};
   buf.data = str_dest ;
   DEV_open(DEV_DEBUG);
   MEM_move(str_src,"2F0F",4);
   BUF_set_counted_str(&buf,str_src,4);
   DBG_CR;
   DBG_STR("buf = ");
   DBG_DUMP(&buf);
   DEV_close(DEV_DEBUG);
}
```

3.1.9 BUF_set_str

Copy the null-terminated string pointed by *str* to the buffer object pointed by *buf*. Raise an XCP_OUT_OF_MEMORY if the length of the string is greater than the size of the destination buffer.

void BUF_set_str (BUF_Handle *buf***, char ****str* **);**

Return Value

none

Parameters

input	*str*	null-terminated string to be copied
output	*buf*	output buffer object

Exceptions

XCP_OUT_OF_MEMORY	*str* length greater than *buf->size*

Example

```
#include <buf.h>
#include <mem.h>
#include <dbg.h>
void app(void)
{
   char str_src[10], str_dest[20];
   BUF_Object buf = {0,20,0};
   buf.data = str_dest ;
   DEV_open(DEV_DEBUG);
   MEM_move(str_src,"2F0F",4);
   str_src[4]=0 ;
   BUF_set_str(&buf,str_src);
   DBG_CR;
   DBG_STR("buf = ");
   DBG_DUMP(&buf);
   DEV_close(DEV_DEBUG);
}
```

3.2 Debug functions

	C function	Main Token invoked
3.2.1	ASSERT	THROW
3.2.2	DBG_ASSERT	THROW
3.2.3	DBG_breakpoint	BREAKPNT
3.2.4	DBG_CR	DEVEMIT
3.2.5	DBG_CSTR	DEVWRITE
3.2.6	DBG_CSTR_ASCII	DEVWRITE
3.2.7	DBG_DUMP	DEVWRITE
3.2.8	DBG_DUMP_ASCII	DEVWRITE
3.2.9	DBG_HEX	DEVWRITE
3.2.10	DBG_INT	DEVWRITE
3.2.11	DBG_LOG_START	
3.2.12	DBG_LOG_STOP	
3.2.13	DBG_STR	DEVWRITE
3.2.14	DBG_STR_CR	DEVWRITE

Table 4: Debug functions and macros

3.2.1 ASSERT

Perform a throw *xcp* if the condition *cond* is not satisfied. The source must be compiled with the DEBUG option.

void ASSERT (bool *cond***, int** *xcp***);**

Return Value

none

Parameters

input	*cond*	condition to satisfy
	xcp	throw code to perform

Example

```
#include <dbg.h>
#define ERR_PROCESSING_ERROR 1

int divide (int x, int y)
{
   ASSERT(y != 0,ERR_PROCESSING_ERROR);
   return x/y;
}
```

3.2.2 DBG_ASSERT

Check the condition *cond* and throw an error code *xcp* after printing *str* on the debug window

if the condition is not satisfied. The source must be compiled with the DEBUG option.

void DBG_ASSERT (int *cond***, char*** *str***, int** *xcp***);**

Return Value

none

Parameters

input	*cond*	condition to satisfy
	xcp	throw code to perform

Example

```
#include <dbg.h>
#define ERR_PROCESSING_ERROR 1

int divide (int x, int y)
{
   ASSERT(y != 0,ERR_PROCESSING_ERROR);
   return x/y;
}
```

3.2.3 DBG_breakpoint

Interrupt the execution after this command. The source must be compiled with the DEBUG option.

void DBG_breakpoint (void);

Return Value

none

Parameters

none

3.2.4 DBG_CR

Print a carriage return on the debug output device. The source must be compiled with the DEBUG option.

DBG_CR;

Return Value

none

Parameters

none

3.2.5 DBG_CSTR

Print *len* characters of the string *str* on the debug output device in hexadecimal mode. The source must be compiled with the DEBUG option.

void DBG_CSTR (char* *str,* **int** *len***);**

Return Value

none

Parameters

input	*str*	string to print
	len	length of the input string

3.2.6 DBG_CSTR_ASCII

Print *len* characters of the string *str* on the debug output device in ascii mode. The source must be compiled with the DEBUG option.

void DBG_CSTR_ASCII (char* *str,* **int** *len***);**

Return Value

none

Parameters

input	*str*	string to print
	len	length of the input string

3.2.7 DBG_DUMP

Print *buf->len* characters of the string *buf->data* on the debug output device in hexadecimal mode. The source must be compiled with the DEBUG option.

void DBG_DUMP (Buf_Handle *buf***);**

Return Value

none

Parameters

input	*buf*	buffer to print

3.2.8 DBG_DUMP_ASCII

Print *buf->len* characters of the string *buf->data* on the debug output device in ASCII mode. The source must be compiled with the DEBUG option.

void DBG_DUMP_ASCII (Buf_Handle *buf***);**

Return Value

none

Parameters

input	*buf*	buffer to print

3.2.9 DBG_HEX

Print the value *x* as an integer on the debug output device in ASCII hexadecimal notation. The source must be compiled with the DEBUG option.

void DBG_HEX (int *x*);

Return Value

none

Parameters

input	*x*	value to print

3.2.10 DBG_INT

Print the value *x* as a signed integer on the debug output device in ASCII decimal notation. The source must be compiled with the DEBUG option.

void DBG_INT (int *x*);

Return Value

none

Parameters

input	*x*	value to print

3.2.11 DBG_LOG_START

This macro definition enables debug logging functions. This is also the default option.

DBG_LOG_START

Return Value

none

Parameters

none

3.2.12 DBG_LOG_STOP

This macro definiton disables the debug logging functions. By default the debug logging functions are enabled.

DBG_LOG_STOP

Return Value

none

Parameters

none

3.2.13 DBG_STR

Print the NULL terminated string *data* on the debug output device. The source must be compiled with the DEBUG option.

void DBG_STR (char ****data*);**

Return Value

none

Parameters

input	*data*	data to print

3.2.14 DBG_STR_CR

Print the NULL terminated string *data* and a carriage return on the debug output device. The source must be compiled with the DEBUG option.

void DBG_STR_CR (char ****data*);**

Return Value

none

Parameters

input	*data*	data to print

3.3 Databases Services

	C function	Main Token invoked
3.3.1	DBS_append	DBADDREC
3.3.2	DBS_avail	DBAVAIL
3.3.3	DBS_define	
3.3.4	DBS_delete_key	DBDELBYKEY
3.3.5	DBS_delete_record	DBDELREC
3.3.6	DBS_direct_read	DBSTRFETCH
3.3.7	DBS_find_key	DBMATCHBYKEY
3.3.8	DBS_get_current	
3.3.9	DBS_init	DBINITIALISE
3.3.10	DBS_make_current	DBMAKECURRENT
3.3.11	DBS_new_key	DBADDBYKEY

Table 5: DBS functions

	C function	Main Token invoked
3.3.12	DBS_new_record	DBADDREC
3.3.13	DBS_read	DBSTRFETCH
3.3.14	DBS_size	DBSIZE
3.3.15	DBS_write	DBSTRSTORE

Table 5: DBS functions (continued)

3.3.1 DBS_append

Add a record at the end of the non ordered database *db* (i.e. at the record number given by `DBS_avail()`). The new record will be initialised with the contents of the buffer pointed by *record*. Exception `XCP_DB_INVALID_FUNCTION` will be thrown if the current database type is read-only, compiled or ordered. Exception `XCP_OUT_OF_MEMORY` will be thrown if there is not enough storage space.

int DBS_append (DBS_Handle *db***, void ****record***);**

Return Value

record number of the appended record

Parameters

input	*db*	address of Database Parameter Block
	record	address of the buffer containing the record data to be written

Exceptions

`XCP_DB_INVALID_FUNCTION`	database is read only,compiled or ordered
`XCP_OUT_OF_MEMORY`	not enough storage space

Example

```
#include <dbs.h>
#include <dbg.h>

typedef struct demo_struct {
   char name[19];
   char id_code;
} demo_db;

void demo_append(DBS_Handle DBS_ptr, demo_db *demo_db_rec)
{
   int rec_nr, xcp;
   xcp = XCP_arg_catch(2,(FUNC)(DBS_append),DBS_ptr,demo_db_rec);
   if (xcp != 0) {
      DBG_STR("Error : Record not added"); DBG_CR;
   }
   else {
      rec_nr = XCP_return_catch;
      DBG_STR("append to demo_db; record number :");
      DBG_INT(rec_nr);
   }
}
```

3.3.2 DBS_avail

Return the record number of the next available record in the database *db*. For an empty database, the number 0 is returned.

int DBS_avail (DBS_Handle *db***);**

Return Value

number of the next available record

Parameters

input	*db*	address of Database Parameter Block

Example

```
#include <dbs.h>
#include <dbg.h>

void demo_avail(DBS_Handle demo_ptr)
{
   int i = DBS_avail(demo_ptr);
   DBG_STR("Number of records for database demo : ");
   DBG_INT(i); DBG_CR;
}
```

3.3.3 DBS_define

Macro definition. Define and add a database parameter block (DPB) to the DBP linked list. The macro is invoked before any function definition. The DPBs are defined in the reverse order, starting with a DPB pointing to `NULL` and ending with `DBS_root` (See also Section 2.2), so that each DPB can be defined before being referenced.

In other words, the last DPB of the list is defined first and the value of its corresponding *next_dbs_pb* is a null-pointer. Each DPB in the list has its *next_dbs_pb* pointer pointing to the DPB defined just before. The last DPB being defined is the first of the list and its *dbs_pb* value is `DBS_root`.

The *data* parameter points to the first of a list of records defined before the `DBS_define` invocation in the program if the database is of type initialized. If not, it must point to `NULL` and the Virtual Machine will assign a value to the pointer at module load time.

The *available* parameter contains the number of records in the data list if the database is of type initialized. If not, it must be zero.

The *key_offset* and *key_size* parameters define the position and size of the unique key in the record. They are not used if the database type is not ordered.

void DBS_define (DBS_pb *dbs_pb***, DBS_Handle** *next_dbs_pb***, void*** *data***,**
unsigned int *type***, unsigned** *rec_size***, unsigned** *available***, unsigned** *key_offset***,**
unsigned *key_size***);**

Return Value

none

Parameters

input	*dbs_pb*	name of the DPB variable being defined, the first
	next_dbs_pb	address of the next DPB variable
	data	pointer to an initialised array of records
	type	characteristics of the database (`DBS_VOLATILE` or `DBS_NON_VOLATILE`, `DBS_INITIALIZED`, `DBS_NOT_ORDERED` or `DBS_ORDERED`, `DBS_RW` or `DBS_RO`
	rec_size	fixed size of the record in the database
	available	number of records in initialised array
	key_offset	offset of the key inside the record
	key_size	size of the key

Exceptions

none

Example

```
#include <dbs.h>

// define a record structure
typedef struct demo_struct {
   unsigned char key[8];
   unsigned char rec_nr;
} demo_db_rec;

// define a database with a 8-character key at the beginning of the record
// the database type is ordered, read-write and volatile
DBS_define(DBS_root,       // last definition of the list
   NULL,                   // only 1 DPB is defined
   0,                      // not initialized
   DBS_VOLATILE|DBS_RW|DBS_ORDERED,
   sizeof(demo_db_rec),    // record size
   0,                      // no records initialized
   0,                      // key in first position
   8);                     // key length cell-aligned
```

Remark

The macro definition must be invoked in the global variable definition section of the program.

3.3.4 DBS_delete_key

Search the database pointed to by *db* for a record whose key matches the string specified by *key* buffer. If the match is successful, the matching record will be deleted. Storage requirements are decreased by this function. The length of the buffer may be shorter than the defined length of the key field for this structure, with the remaining characters being compared to space (ASCII 0x20) characters. `XCP_DB_INVALID_FUNCTION` will be thrown if the current database type is not ordered, or if it is read-only or compiled.

int DBS_delete_key (DBS_Handle *db***, BUF_Handle** *key***);**

Return Value

-1 if key found and record deleted, otherwise 0

Parameters

input	*db*	pointer to the Database Parameter Block
	key	search and match key

Remarks

- If the length of the key is less than that of the key field, the key is right-filled with blank characters for matching and record deletion.
- If the length of the key is larger than that of the key field, it is right-truncated to that length for matching and record deletion.

Exceptions

`XCP_DB_INVALID_FUNCTION`	database is read only, database is not ordered or database is a compiled database

Example

```
#include <dbs.h>
#include <dbg.h>
#include <buf.h>

DBS_Handle DBS_ptr;
BUF_Handle key;
int i;
...
i = DBS_delete_key(DBS_ptr, key);
if ( i != -1 ) {
   DBG_STR("After  DBS_delete_key test_db: ");
   DBG_INT(i);
}
else {
   DBG_STR("Error: does not Exist !!");
}
```

3.3.5 DBS_delete_record

Delete a record specified by its record number from the database pointed by *db*. Storage requirements are decreased by this function.
Exception `XCP_DB_INVALID_FUNCTION` will be thrown if the current database type is read-only or compiled. Exception `XCP_DB_INVALID_RECORD` will be thrown if the current record number is not within the range.

void DBS_delete_record (DBS_Handle *db***, unsigned int** *record_nr***);**

Return Value

none

Parameters

input	*db*	pointer to the database parameter block

	record_nr	number of the record to delete (first record is 0)

Exceptions

XCP_DB_INVALID_FUNCTION	database is read only or database is a compiled database
XCP_DB_INVALID_RECORD	record number not within the range

Example

```
#include <dbs.h>

DBS_Handle demo_ptr;
// delete first record
DBS_delete_record(demo_ptr, 0);
```

3.3.6 DBS_direct_read

Read record *rec_nr* in *db* database and return the address pointing to the record buffer of the database. Note that the function provides direct access to the record buffer (See Volume 1, Section 4.3) and that no memory has to be allocated by the calling function..

int DBS_direct_read (DBS_Handle *db***,unsigned** *rec_nr***, void**** *buf***);**

Return Value

-1 if record read successfully, 0 if *rec_nr* is out of range

Parameters

input	*db*	pointer to the database parameter block
	rec_nr	number of the record
output	*buf*	pointer to the address of the record buffer

Example

```
#include <dbs.h>
#include <dbg.h>
#include <xcp.h>
#include <mem.h>
#include <dbg.h>
#include <str.h>
#include <buf.h>
#define LEGEND_SIZE 16

typedef struct {
   char key[3];
   char legend[LEGEND_SIZE];
   char recnr;
} legend_rec;

legend_rec recs[] = {
   {"123","legend first rec",'1'},
   {"456","legend secnd rec",'2'},
   {"789","legend first rec",'3'},
   {"ABC","legend secnd rec",'4'}
};
```

```
DBS_define(DBS_root,
   0,
   recs[0].key,
   DBS_INITIALIZED,     // compiled DB
   sizeof (legend_rec), // size of each record
   4,                   // compiled, 4 records
   0,                   // key_offset
   0                    // key length
);

void app(void)
{
   legend_rec *rec;
   char temp_str[20];
   BUF_Object buf= {0,20,0} ;
   buf.data= temp_str ;
   DEV_open(DEV_DEBUG);
   DBG_STR_CR("DBS_direct_read");
   DBS_direct_read(&DBS_root,1,(void**)&rec);
   BUF_set_counted_str(&buf,rec->key,3);
   DBG_STR("Key must be 456:");DBG_DUMP_ASCII(&buf);DBG_CR;
   BUF_set_counted_str(&buf,rec->legend,LEGEND_SIZE);
   DBG_STR("legend must be legend secnd rec:");
   DBG_DUMP_ASCII(&buf);DBG_CR;
   BUF_set_counted_str(&buf,&rec->recnr,1);
   DBG_STR("record number must be 2:");DBG_DUMP_ASCII(&buf);DBG_CR;
   DEV_close(DEV_DEBUG);
}
```

3.3.7 DBS_find_key

Search the database *db* for a match on the key field against the string specified by the *key* buffer. Key length may be shorter than the defined length of the key field for this structure, with the remaining characters being compared to space (ASCII 0x20) characters. Exception `XCP_DB_INVALID_FUNCTION` will be thrown if the current database type is not ordered.

int DBS_find_key (DBS_Handle *db***, BUF_Handle** *key***);**

Return Value

number of the matching record (first record = 0), –1 if no match found

Parameters

input	*db*	pointer to the Database Parameter Block
	key	value of the key to be found

Remarks

- If the length of the key is less than that of the key field, the key is filled with blank characters for matching.
- If the length of the key is larger than that of the key field, it is truncated to that length for matching.

Exceptions

`XCP_DB_INVALID_FUNCTION` database is not ordered

Example

```
#include <dbs.h>
#include <dbg.h>

int i;
DBS_Handle demo_ptr;
BUF_Object key = { 11, 12, 0 };
key.data = "TEST-INDEXB";
i = DBS_find_key(demo_ptr, &key);
if ( i != -1 ) {
   DBG_STR("number of the matching record: ");
   DBG_INT(i); DBG_CR;
}
else {
   DBG_STR("Error: Already Exist!!"); DBG_CR;
}
```

3.3.8 DBS_get_current

Retrieve the database currently selected. To be used in a function accessing a database defined in a Forth module.

DBS_Handle DBS_get_current (void);

Return Value

pointer to the currently selected database.

Parameters

none

Example

```
// get the size of the current database
#include <dbs.h>
int i;
i = DBS_size ( DBS_get_current());
DBG_INT(i);
```

3.3.9 DBS_init

Delete all records in the database *db*, erasing their data, and set “available” record number of the database to zero. Exception `XCP_DB_INVALID_FUNCTION` will be thrown if the current database type is read-only or compiled.

void DBS_init (DBS_Handle *db***);**

Return Value

none

Parameters

input	*db*	pointer to the Database Parameter Block

Exceptions

`XCP_DB_INVALID_FUNCTION`	database is read-only or compiled

Example

```
#include <dbs.h>

DBS_Handle demo_ptr;
DBS_init(demo_ptr);
```

3.3.10 DBS_make_current

Make the database *db* current. To be used before calling a function which accesses a database from a Forth module.

void DBS_make_current(DBS_Handle *db***);**

Return Value

none

Parameters

input	*db*	pointer to the Database Parameter Block

Example

```
#include <dbs.h>

DBS_Handle demo_ptr;
...
DBS_make_current(&demo_ptr);
```

3.3.11 DBS_new_key

Search the database pointed by *db* for a match on the key field against the string specified by the *key* buffer. If the match is successful, the function returns -1. If the match is not successful, a new record will be inserted at the correct position in the database and the position of the new record will be returned. The content of the new record is unspecified except for its key field, which will contain the given key. The length of the *key* buffer may be shorter than the defined length of the key field for this structure, with the remaining characters being compared to space (ASCII 0x20) characters. Exception `XCP_DB_INVALID_FUNCTION` will be thrown if the current database type is nor ordered, or if it is read-only or compiled. Exception `XCP_OUT_OF_MEMORY` will be thrown if there is not enough storage space.

int DBS_new_key (DBS_Handle *db***, BUF_Handle** *key***);**

Return Value

the record number of the new record, -1 if the key already exists

Parameters

input	*db*	pointer to the Database Parameter Block
	key	value of the key

Exceptions

`XCP_DB_INVALID_FUNCTION`	database is read only, database is a compiled database or database is not ordered
`XCP_OUT_OF_MEMORY`	not enough storage space

Remark

- If the length of the key is less than that of the key field, the key is filled with blank characters for matching and record insertion. If the length of the key is larger than that of the key field, it is truncated to that length for matching and record insertion.

Example

```
#include <dbs.h>
#include <dbg.h>
// ... for an example of database definition, see above DBS_define
int i;
DBS_Handle demo_ptr;
BUF_Object key = { 8, 8, 0 };
key.data = "12345678";
i = DBS_new_key(demo_ptr, &key);
if ( i != -1 ) {
   DBG_STR("number of the new record: ");
   DBG_INT(i); DBG_CR;
}
else {
   DBG_STR("Key Already Exists!!"); DBG_CR;
}
```

3.3.12 DBS_new_record

Add a record at the end of the database pointed by *db* (i.e. at the record number given by `DBS_avail()`). The content of the new record is unspecified. Storage requirements are increased by this function, and it may cause exception `XCP_OUT_OF_MEMORY` to be thrown. Exception `XCP_DB_INVALID_FUNCTION` will be thrown if the current database type is read-only, compiled or ordered.

int DBS_new_record (DBS_Handle *db***);**

Return Value

number of the added record (first record is 0)

Parameters

input	*db*	pointer to the Database Parameter Block

Exceptions

`XCP_DB_INVALID_FUNCTION`	database is read only, database is a compiled database, or ordered

XCP_OUT_OF_MEMORY	not enough storage space

Example

```
#include <dbs.h>
#include <dbg.h>

int i;
i = DBS_new_record(DBS_ptr);
DBG_STR("DBS_new_record nr =");
DBG_INT(i);
```

3.3.13 DBS_read

Read the data in the database pointed by *db* at the record number *rec_nr* and store it in the buffer pointed by *record*. The buffer must be large enough to hold the record read from the database. The function returns -1 if the record was read successfully and 0 if *record_nr* is not within the range of existing records in the database.

int DBS_read (DBS_Handle *db***, unsigned** *record_nr***, void ***record***);**

Return Value

-1 if the record was read successfully, 0 if *record_nr* is out of range

Parameters

input	*db*	address of database parameter block
	record_nr	number of record to read (first record is 0)
output	*record*	buffer where the record should be written

Example

```
#include <dbs.h>
#include <dbg.h>
#include <mem.h>

DBS_Handle  DEMO_DB_DPB(void);
typedef struct demo_struct {
   char name[19];
   char rec_id;
} demo_db;

void demo_read(DBS_Handle DBS_ptr, int rec_number, demo_db *demo_db_rec)
{
   char msg[32];
   MEM_move(demo_db_rec->name,"AAA naming",10);
   if (DBS_read(DBS_ptr,rec_number,demo_db_rec) != 0) {
      MEM_move(msg, demo_db_rec->name, 19);
      msg[19] = 0;
      DBG_STR(msg); DBG_STR(" : ");
      DBG_INT(demo_db_rec->rec_nr);
   }
   else {
      DBG_STR("##Error");
   }
}
```

3.3.14 DBS_size

Return the length of a record in the database *db*.

int DBS_size (DBS_Handle *db***);**

Return Value

the length of the records in the database

Parameters

input	*db*	pointer to the Database Parameter Block

Example

```
#include <dbs.h>
#include <dbg.h>

int i;
DBS_Handle demo_ptr;
i = DBS_size(demo_ptr);
DBG_STR("Record size = ");
DBG_INT(i);
```

3.3.15 DBS_write

Store `DBS_size(db)` (record size) bytes of the byte sequence at *record* in the database *db* at the record number *record_nr*. The record must have been created using `DBS_new_key`, `DBS_new_record` or `DBS_append`. The function returns -1 if the record is written successfully and 0 if *record_nr* is not within the range of existing records in the database. Exception `XCP_DB_INVALID_FUNCTION` will be thrown if the current database type is read-only or compiled.

int DBS_write (DBS_Handle *db***, unsigned** *record_nr***, void ****record***);**

Return Value

-1 if the record is written successfully, 0 if the *record_nr* is out of range

Parameters

input	*db*	address of database parameter block
	record_nr	number of record to write (first record is 0)
	record	buffer with the data to be written to the record

Exceptions

`XCP_DB_INVALID_FUNCTION`	database is read only.

Example

```
#include <dbs.h>
#include <dbg.h>
#include <str.h>

typedef struct demo_struct{
   char name[19];
```

```
   char id;
} demo_db;

void demo_write(DBS_Handle DBS_ptr, unsigned rec_nr, demo_db *demo_db_rec)
{
   char msg[32];
   MEM_move(demo_db_rec->name, "AAA naming", 10);
   if (DBS_write(DBS_ptr,rec_nr,demo_db_rec)) {
      MEM_move(msg, demo_db_rec->name, 19);
      msg[19] = 0;
      DBG_STR(msg); DBG_STR(" : ");
      DBG_INT(demo_db_rec->id);
   }
   else {
      DBG_STR("Error");
   }
};
```

3.4 Devices and I/O Services

	C function	Main Token emulated or Forth function
3.4.1	DEV_beep	BEEP (Forth)
3.4.2	DEV_centred_type	CENTRED_TYPE (Forth)
3.4.3	DEV_close	DEVCLOSE
3.4.4	DEV_connect	DEVCONNECT
3.4.5	DEV_cr	CR (Forth)
3.4.6	DEV_emit	DEVEMIT
3.4.7	DEV_hangup	DEVHANGUP
3.4.8	DEV_ioctl	DEVIOCTL
3.4.9	DEV_key	DEVEKEY
3.4.10	DEV_key_pressed	DEVEKEYQ
3.4.11	DEV_locate	DEVATXY
3.4.12	DEV_open	DEVOPEN
3.4.13	DEV_output	DEVOUTPUT
3.4.14	DEV_page	PAGE (Forth)
3.4.15	DEV_read	DEVREAD
3.4.16	DEV_space	SPACE (Forth)
3.4.17	DEV_spaces	SPACES (Forth)
3.4.18	DEV_status	DEVSTATUS

Table 6: DEV functions

	C function	Main Token emulated or Forth function
3.4.19	DEV_timedread	DEVTIMEDREAD
3.4.20	DEV_type	TYPE (Forth)
3.4.21	DEV_type_int	
3.4.22	DEV_write	DEVWRITE

Table 6: DEV functions (continued)

3.4.1 DEV_beep

Send a beep to the specified device *dev.*

int DEV_beep (DEV_Handle *dev***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number

Example

```
#include <dev.h>
void app(void)
{
   // You'll hear one beep
   DEV_open(DEV_DISPLAY);
   DEV_beep(DEV_DISPLAY);
   DEV_close(DEV_DISPLAY);
}
```

3.4.2 DEV_centred_type

Position the display cursor so that *buf->len* characters will be centred in a field of *length* characters from the current cursor position. If *buf->len* is greater than zero, send the character string specified by *buf* to the display.

int DEV_centred_type (DEV_Handle *dev***, BUF_Handle** *buf***, int** *length***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number
	buf	data to print
	length	the centring length

Example

```
#include <dev.h>
void app(void)
{
   BUF_Object buf = { 0, 10, 0 };
   int length = 16;
   buf.data = "test";
   buf.len = 4;
   DEV_open(DEV_DISPLAY);
   DEV_locate(DEV_DISPLAY, 0, 0);
   // results looks like
   //  1234567890123456
   //        test
   DEV_centred_type(DEV_DISPLAY, &buf, length);
   DEV_close(DEV_DISPLAY);
}
```

3.4.3 DEV_close

Close the given device *dev.* After closing the device, the device is not accessible anymore and all the device access functions applied on a closed device will return ior -32759 (XCP_DEV_MUST_BE_OPENED).

int DEV_close (DEV_Handle *dev***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number

Example

```
See other device services examples.
```

3.4.4 DEV_connect

Connect to remote system using device *dev. buf* contains the access parameters (for example, telephone number plus modem command characters).
As long as the connection is not established, DEV_status applied on device *dev* returns ior -31720 (XCP_MDM_CONNECTION_IN_PROG). If the connection is successfully established, DEV_status returns ior 0. Every other ior indicates an error.

int DEV_connect (DEV_Handle *dev***, BUF_Handle** *buf***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number

	buf	string containing the dial parameter (phone number including special wait characters and Pulse mode eventually (the string following the Hayes ATD command))

Example

```
#include <dev.h>
void app(void)
{
   // Establish a modem connection using the phone number 003223525111
   // dial 0 to get an external line, wait one second (",") and dial
   // number. If a P is added as the first character ("P0,003223525111"),
   // the modem uses the pulse mode to dial.
   BUF_Object buf = { 12, 13, 0 };
   buf.data = "0,003223525111";
   DEV_open(DEV_MODEM);
   if (DEV_connect(DEV_MODEM, &buf) != 0) {
      DEV_type(DEV_DISPLAY," No Connection");
   };
}
```

3.4.5 DEV_cr

Send a carriage return to the specified device *dev.*

int DEV_cr (DEV_Handle *dev***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number

Example

```
#include <dev.h>
#include <dbg.h>

void app(void)
{
   DEV_open(DEV_DEBUG);
   DBG_STR("Line 1");
   DEV_cr(DEV_DEBUG);
   DBG_STR("Line 2");
   DEV_close(DEV_DEBUG);
}
```

3.4.6 DEV_emit

Transmit one character to the given output device *dev.*

int DEV_emit (DEV_Handle *dev***, char** *c***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number
	c	character to be sent

Example:

```
#include <dev.h>
void app(void)
{
   // The character 'c' is sent to the display.
   DEV_open(DEV_DISPLAY);
   DEV_emit(DEV_DISPLAY, 'c');
   DEV_close(DEV_DISPLAY);
}
```

3.4.7 DEV_hangup

End the current modem session on the given device *dev.*

int DEV_hangup (DEV_Handle *dev***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number

Example

```
#include <dev.h>
void app(void)
{
   DEV_open(DEV_MODEM);
   ...
   if (DEV_hangup(DEV_MODEM) != 0) {
      DEV_type(DEV_DISPLAY,"Hangup Failed");
   }
   DEV_close(DEV_MODEM);
}
```

3.4.8 DEV_ioctl

Perform the I/O control function *fn* on device *dev* with *len* integer arguments in the array *args*. Individual operations are device dependent and are defined against supported devices in Appendix B. Ior -21 (`XCP_UNSUPPORTED_OPERATION`) will be returned when in a particular implementation device *dev* is supported but function *fn* is not.

int DEV_ioctl (DEV_Handle *dev***, int** *fn***, int** *len***, int** *args[]***);**

Return Value

0 if no error, otherwise a device and function dependent ior (see Appendix B)

Parameters

input	*dev*	device number

fn	the function to execute
len	the number of arguments in *args*
args	the arguments used in function

Example

```
#include<dev.h>
void app(void)
{
   int args;
   DEV_open(DEV_MODEM);
   // Set Time out for the modem device.
   args = 3000;
   if (DEV_ioctl(DEV_MODEM,DEV_FN_TIME_OUT,1,&args)!= 0) {
      DEV_type(DEV_DISPLAY,"ERTime");
   }
   DEV_close(DEV_MODEM);
}
```

3.4.9 DEV_key

Read an extended character from input device *dev.* This function does not take into account the time-out set by the `DEV_ioctl` function. It waits for ever until an extended character is received.

int DEV_key (DEV_Handle *dev***);**

Return Value

return the extended character

Parameters

input	*dev*	device number

Exceptions

Besides the normal throw codes listed in Table 23 and Table 24, this function also throws the device dependent ior's from Table 25 that are normally returned as ior.

Example

```
#include <dev.h>
void app(void)
{
   int key_nr;
   DEV_open(DEV_KEYBOARD);
   // get key from keyboard if a key has been pressed on the keyboard
   if (DEV_key_pressed(DEV_KEYBOARD)== -1) {
      key_nr = DEV_key(DEV_KEYBOARD);
      if ((key_nr &0xFF00) != 0x00) {
         //  ... special function key
      }
      else {
         // ... numbers ...
      }
   }
   else {
```

```
      DEV_type(DEV_DISPLAY," Dev not ready");
   };
   DEV_close(DEV_DISPLAY);
}
```

3.4.10 DEV_key_pressed

Return -1 if a character is ready to be read from the input device *dev.*

int DEV_key_pressed (DEV_Handle *dev***);**

Return Value

-1 if a character is present on the device, otherwise 0

Parameters

input	*dev*	device number

Exceptions

Besides the normal throw codes listed in Table 23 and Table 24, this function also throws the device dependent ior's from Table 25 that are normally returned as ior.

Example

```
#include <dev.h>
void app(void)
{
   // Print "dev ready" on the terminal display if a key has been pressed
   // on the keyboard
   DEV_open(DEV_KEYBOARD);
   DEV_open(DEV_DISPLAY);
   if (DEV_key_pressed(DEV_KEYBOARD)== -1) {
      DEV_type(DEV_DISPLAY," Dev ready");
   }
   else {
      DEV_type(DEV_DISPLAY," Dev Error");
   }
   DEV_close(DEV_KEYBOARD);
   DEV_close(DEV_DISPLAY);
}
```

3.4.11 DEV_locate

Position the cursor to co-ordinates (x, y) on device *dev.* (0,0) is upper left corner of the display screen.

int DEV_locate (DEV_Handle *dev***, int** *x***, int** *y***);**

Return Value

0 if no error, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number
	x	column position (first column is 0)
	y	line position (first line is 0)

Example

```
#include <dev.h>
void app(void)
{
   // The text "Hello World" is written on the terminal
   // at the beginning of the second line.
   DEV_open(DEV_DISPLAY);
   DEV_locate(DEV_DISPLAY, 0, 1);
   DEV_type(DEV_DISPLAY,"Hello World");
   DEV_close(DEV_DISPLAY);
}
```

3.4.12 DEV_open

Open device *dev*. On buffered devices `DEV_open` clears all data from the buffers. Ior -32759 (`XCP_DEV_MUST_BE_OPENED`) will be returned by all device access functions applied on a device that is not open. All devices are closed by start-up of the terminal. Applying the `DEV_open` function on a device that is already open, results in a no-op and an ior -32758 (`XCP_DEV_ALREADY_OPENED`).

int DEV_open (DEV_Handle *dev***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number

Example

See other device services examples.

3.4.13 DEV_output

Pass the function *fn* to device *dev* for execution and return zero if the function was accepted for processing. The function fn must be a void function which takes a device number as parameter.

Depending upon the implementation, execution of *fn* may proceed concurrently with further processing in the function which called `DEV_output`.

Any non caught exception which occurs during execution of *fn* will cause *fn* to terminate immediately, freeing the device for further use.

Programs can use `DEV_status` to determine whether the execution of *fn* has completed.

int DEV_output (DEV_Handle *dev***, void (***fn***)(DEV_Handle** *dev***));**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number
	fn	address of the function to be passed

Example

```
#include <dev.h>

void print_ticket(DEV_Handle dev)
{
   char *str;
   if ( dev == DEV_DISPLAY ) {
      str = "DTicket";
   }
   else {
      str = "PTicket";
   }
   DEV_type(dev,str);
}

void app(void)
{
   DEV_open(DEV_SYSTEM_PRINTER);
   DEV_open(DEV_DISPLAY);
   if (DEV_output(DEV_SYSTEM_PRINTER, print_ticket) != 0) {
      DEV_type(DEV_DISPLAY,"ER-output");
   };
   // same test as above with change of device.
   if (DEV_output(DEV_DISPLAY, print_ticket) != 0) {
      DEV_type(DEV_DISPLAY,"ER-output");
   };
   DEV_close(DEV_SYSTEM_PRINTER);
   DEV_close(DEV_DISPLAY);
}
```

3.4.14 DEV_page

Send a Form Feed to the specified device *dev.*

int DEV_page (DEV_Handle *dev***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number

Example

```
#include <dev.h>
#include <dbg.h>
void app(void)
{
   // Look at the printer.
   DEV_open(DEV_SYSTEM_PRINTER);
```

```
    DEV_type(DEV_SYSTEM_PRINTER, "Line 1");
    DEV_page(DEV_SYSTEM_PRINTER);
    DEV_type(DEV_SYSTEM_PRINTER, "Line 2");
    DEV_close(DEV_SYSTEM_PRINTER);
}
```

3.4.15 DEV_read

Read *buf.len* bytes from input device *dev*, returning a device dependent ior. The size of an element within the *buf.len* bytes is device dependent. For example, on a keyboard each element is two bytes, an extension code in the first byte and the key value in the second byte. This function does not take into account the time-out set by the `DEV_ioctl` function. It waits until *buf.len* characters are received. If one wants to make use of the time-out value, `DEV_timed_read` should be used.

int DEV_read (DEV_Handle *dev***, BUF_Handle** *buf***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number
output	*buf*	reception buffer

Example

```
#include <dev.h>
#include <buf.h>
#include <dbg.h>
#include <str.h>
void app(void)
{
   // Three characters are read from the keyboard and printed.
   BUF_Object buf = {0,20,0};
   int result;
   char char_tempo[20];
   buf.data = char_tempo;
   buf.size = 20;
   buf.len = 6;
   DEV_open(DEV_KEYBOARD);
   DEV_open(DEV_DEBUG);
   if ((result = DEV_read(DEV_KEYBOARD,&buf))==0) {
      DBG_DUMP(&buf);
      DBG_STR(" YES");
   }
   else {
      DBG_STR(" No");
   }
   DEV_close(DEV_KEYBOARD);
   DEV_close(DEV_DEBUG);
}
```

3.4.16 DEV_space

Send a space to the specified device *dev*.

int DEV_space (DEV_Handle *dev***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number

Example

```
#include <dev.h>
#include <dbg.h>
void app(void)
{
   // Look at the printer.
   DEV_open(DEV_SYSTEM_PRINTER);
   DEV_type(DEV_SYSTEM_PRINTER, "Line 1");
   DEV_type(DEV_SYSTEM_PRINTER, "Line 2");
   DEV_cr(DEV_SYSTEM_PRINTER);
   DEV_type(DEV_SYSTEM_PRINTER, "Line 1");
   DEV_space(DEV_SYSTEM_PRINTER);
   DEV_type(DEV_SYSTEM_PRINTER, "Line 2");
   DEV_close(DEV_SYSTEM_PRINTER);
}
```

3.4.17 DEV_spaces

Send *number* spaces to the specified device *dev.*

int DEV_spaces (DEV_Handle *dev***, int** *number***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number
	number	the number of spaces to send to the device

Example

```
#include <dev.h>
#include <dbg.h>
void app(void)
{
   // Look at the printer
   DEV_open(DEV_SYSTEM_PRINTER);
   DEV_type(DEV_SYSTEM_PRINTER, "Line 1");
   DEV_type(DEV_SYSTEM_PRINTER, "Line 2");
   DEV_cr(DEV_SYSTEM_PRINTER);
   DEV_type(DEV_SYSTEM_PRINTER, "Line 1");
   DEV_spaces(DEV_SYSTEM_PRINTER,5);
   DEV_type(DEV_SYSTEM_PRINTER, "Line 2");
   DEV_close(DEV_SYSTEM_PRINTER);
}
```

3.4.18 DEV_status

Return the status ior of device *dev*, where in the general case “ready” and “serviceable” is indicated by zero and “not ready” is indicated by any other value. A specific device may return non-zero ior values that have significance for that device, as detailed in Appendix B. If the device is currently occupied because it has been selected by a previous execution of the `DEV_output` function, `DEV_status` shall return the standard ior code for “device busy” (see Appendix B) until execution of the procedure passed to `DEV_output` completes.

int DEV_status (DEV_Handle *dev***);**

Return Value

if the device is ready for use, the value returned is zero; any other value is device-dependent (see Appendix B)

Parameters

input	*dev*	device number

Example

```
#include <dev.h>
void app(void)
{
   DEV_open(DEV_DISPLAY);
   if (DEV_status(DEV_DISPLAY) == 0) {
      DEV_type(DEV_DISPLAY," Ok stat");
   }
   DEV_close(DEV_DISPLAY);
}
```

3.4.19 DEV_timed_read

Read *buf->len* bytes from device *dev*, returning a device dependent ior. An ior of zero means success, and any other value is both device and implementation dependent. The size of an element within the string is device dependent. On return *buf->len* gives the actual length of the string read. This length may be less than the input length of the buffer. If the requested number of elements are not received within the specified period of time, the function returns with ior -32766 (time-out). A time-out of 0 will cause the function to return immediately. If the Virtual Machine has buffered enough data, the function will return immediately with ior 0.

int DEV_timed_read (DEV_Handle *dev***, BUF_Handle** *buf***);**

Return Value

0 if all the bytes are successfully read, `XCP_DEV_TIMEOUT` if the number of elements are not received within the specified period of time and in all other cases a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number
output	*buf*	reception buffer

Example

```
#include <dev.h>
#include <buf.h>
#include <dbg.h>
// Three characters are read from the keyboard within 5 seconds
void app(void)
{
   int args = 5000;
   char char_tempo[20];
   BUF_Object buf = { 6, 20, 0 };
   buf.data = char_tempo;
   DEV_open(DEV_KEYBOARD);
   DEV_open(DEV_DEBUG);
   DEV_ioctl(DEV_KEYBOARD, DEV_FN_TIME_OUT, 1, &args);
   if (DEV_timed_read(DEV_KEYBOARD,&buf) == 0) {
      DBG_STR(" Characters inserted are : ");
      DBG_DUMP(&buf);
   }
   else {
      DBG_STR(" Too late");
   }
   DEV_close(DEV_KEYBOARD);
   DEV_close(DEV_DEBUG);
}
```

3.4.20 DEV_type

Display the null-terminated string *str* on the specified device *dev* at the current cursor position.

int DEV_type (DEV_Handle *dev***, char*** *str***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number
	str	null terminated input string

Example

```
#include <dev.h>
void app(void)
{
   DEV_open(DEV_DISPLAY);
   DEV_type(DEV_DISPLAY,"Hello World");
   DEV_close(DEV_DISPLAY);
}
```

3.4.21 DEV_type_int

Display the ASCII string corresponding to the value of *val* on the specified device *dev* at the current cursor position.

int DEV_type_int (DEV_Handle *dev***, int** *val***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number
	val	integer to print

Example

```
#include <dev.h>
void app(void)
{
   int x = 10;
   DEV_open(DEV_DISPLAY);
   DEV_type_int(DEV_DISPLAY,x);
   DEV_close(DEV_DISPLAY);
}
```

3.4.22 DEV_write

Write *buf->len* bytes to the specified output device *dev*, returning a device dependent ior.

int DEV_write (DEV_Handle *dev***, BUF_Handle** *buf***);**

Return Value

0 if *buf->len* bytes were successfully written, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev*	device number
	buf	input buffer

Example

```
#include <dev.h>
#include <buf.h>
void app(void)
{
   BUF_Object buf = { 4, 5, 0 };
   buf.data = "Test";
   DEV_open(DEV_DISPLAY);
   if (DEV_write(DEV_DISPLAY,&buf) == 0) {
      DEV_type(DEV_DISPLAY," Ok");
   }
   else {
      DEV_type(DEV_DISPLAY," Error");
   }
   DEV_close(DEV_DISPLAY);
}
```

3.5 Hot Card List Management services

	C function	Main Token invoked
3.5.1	HCF_delete	hotdelete
3.5.2	HCF_erase	hotinit
3.5.3	HCF_find	hotfind
3.5.4	HCF_insert	hotadd

Table 7: Hot Card List functions

3.5.1 HCF_delete

Delete the entry *buf* from the hot card file.

int HCF_delete (BUF_Handle *buf***);**

Return Value

If a match is found and the entry deleted, the value returned is -1; if a match is not found, the value returned is zero.

Parameters

input	*buf*	A buf object with the data to remove from the list

Example:

```
#include <hcf.h>
#include <buf.h>
#include <dev.h>

void app(void)
   BUF_Object buf_HOT = { 11, 11, 0  };
   buf_HOT.data = " 234567890";
   DEV_open(DEV_DISPLAY);
   if (HCF_delete(&buf_HOT) == -1) {
      DEV_type(DEV_DISPLAY,"Deleted");
   }
   else {
      DEV_type(DEV_DISPLAY,"Error");
   }
   DEV_close(DEV_DISPLAY);
}
```

3.5.2 HCF_erase

Initialise the hot card file to an empty state.

void HCF_erase (void);

Return Value

none

Parameters

none

Example:

```
#include <hcf.h>
HCF_erase();
```

3.5.3 HCF_find

Search the hot card file for an entry matching the entry given by *buf*.

int HCF_find (BUF_Handle *buf***);**

Return Value

-1 if a match is found, otherwise 0

Parameters

input	*buf*	buffer object containing the data to be found in the list

Example:

```
#include <hcf.h>
#include <buf.h>
#include <dev.h>

BUF_Object buf_HOT = { 11, 11, 0 };
buf_HOT.data = " 234567890";
if (HCF_find(&buf_HOT) == -1) {
   DEV_type(DEV_DISPLAY,"Found !");
}
else {
   DEV_write(DEV_DISPLAY,"Error");
}
```

3.5.4 HCF_insert

Insert an entry given by *buf*.

int HCF_insert (BUF_Handle *buf***);**

Return Value

-1 if a match is found and the insertion is done, otherwise 0

Parameters

input	*buf*	buffer object containing the data to insert in the list

Example:

```
#include <hcf.h>
#include <buf.h>
#include <dev.h>

BUF_Object buf_HOT = { 11, 11, 0};
buf_HOT.data = " 234567890";
```

```
if (HCF_insert(&buf_HOT) == -1) {
   DEV_type(DEV_DISPLAY,"Inserted");
}
else {
   DEV_write(DEV_DISPLAY,"Error");
}
```

3.6 Devices and I/O Services - Integrated Circuit Card functions

	C function	Main Token invoked
3.6.1	ICC_off	CARDOFF
3.6.2	ICC_on	CARDON
3.6.3	ICC_order	CARD
3.6.4	ICC_present	CARDABSENT

Table 8: ICC functions

3.6.1 ICC_off

Power off the ICC in the card reader *dev_icc*.

int ICC_off (DEV_Handle *dev_icc***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev_icc*	device handle of the card reader

Example

See `ICC_order()`.

3.6.2 ICC_on

Apply power to the ICC in the card reader *dev_icc* and execute a card reset function. The "Answer to Reset" message shall be returned in the buffer *buf*.

int ICC_on (DEV_Handle *dev_icc***, BUF_Handle** *buf***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*dev_icc*	device handle of the card reader.
output	*buf*	the status bytes.

Example

See `ICC_order()`.

3.6.3 ICC_order

Send the data in the buffer *to_card* to the card in the card reader *dev_icc*, and receive data in *from_card*. The returned *from_card->len* gives the actual length of the string received. The two buffers may share the same memory region.

The buffer *from_card* must provide adequate space for the answer from the card plus two status bytes containing SW1 and SW2. The format of the input and output buffer is the command response APDU format as specified in ISO/IEC 7816-4.

int ICC_order (DEV_Handle *dev_icc***, BUF_Handle** *to_card***, BUF_Handle** *from_card***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*to_card*	command sent to the card
	dev_icc	device handle of the card reader
output	*from_card*	reception buffer : answer from the card

Example

```
#include <icc.h>
#include <dev.h>
#include <dbg.h>
#include <buf.h>


void app(void)
{
   BUF_Object buf_to = { 20, 25, 0 };
   BUF_Object buf_from = { 0, 50, 0 };
   char char_tempo[50];
   char command[25] = {
      0x00,0xA4,0x04,0x00,0x0E,0x31,0x50,0x41,0x59,0x2E,
      0x53,0x59,0x53,0x2E,0x44,0x44,0x46,0x30,0x31,0x00,
      0,0,0,0,0
   };

   // Open the different devices
   DEV_open(DEV_ICC_1);
   DEV_open(DEV_DEBUG);

   // Verify if an IC card is present in the reader
   switch ( ICC_present(DEV_ICC_1)) {
      case DEV_OPERATION_SUCCESSFUL :
        DBG_STR("Present"); DBG_CR;
        break;
      case XCP_DEV_NOT_PRESENT :
         DBG_STR("Device not responding"); DBG_CR;
         break;
```

```
      case XCP_ICC_NOT_PRESENT :
         DBG_STR("No card in reader"); DBG_CR;
         break;
      default :
          break;
   }

   // Power-on the IC card
   buf_from.data = char_tempo;
   if (ICC_on(DEV_ICC_1, &buf_from)==0) {
      DBG_DUMP(&buf_from);
   }
   else {
      DBG_STR(" Error on"); DBG_CR;
   };

   // ICC_order: the reponse of the card must be :
   // 6F 15 84 E 31 50 41 59 2E 53 59 53 2E 44 44 46 30 31 A5 3 88 1 1 90 0
   buf_to.data = command;
   DBG_DUMP(&buf_to);
   buf_from.len = 0;
   ICC_order( DEV_ICC_1, &buf_to, &buf_from);
   DBG_DUMP(&buf_from) ;

   // Power-off the card
   ICC_off(DEV_ICC_1);

   // Close the devices
   DEV_close(DEV_ICC_1);
   DEV_close(DEV_DEBUG);
}
```

3.6.4 ICC_present

Notify the user if a card is present in the card reader *dev_icc*.

int ICC_present (DEV_Handle *dev_icc***);**

Return Value

`XCP_ICC_NOT_PRESENT` if the card is absent, otherwise 0

Parameters

input	*dev_icc*	device handle of the card reader

Remark

This function must not be used to test the presence of a card, it only detects its absence. To detect wether a card is present, `ICC_on()` must be used.

Example

See `ICC_order()`.

3.7 Devices and I/O Services - Magnetic Stripe Functions

	C function	Main Token invoked
3.7.1	MAG_read	MAGREAD
3.7.2	MAG_write	MAGWRITE

Table 9: MAG functions

3.7.1 MAG_read

Read one or more ISO magstripes. The parameter *tracks* is the ISO identifier of magstripe(s) to read, as shown in Table 10. Refer to ISO Standard 7813 for a description of the format of this data.

Track	Magstripes to read
1	ISO1
2	ISO2
3	ISO3
12	ISO1 and ISO2
13	ISO1 and ISO3
23	ISO2 and ISO3
123	ISO1, ISO2 and ISO3

Table 10: ISO parameter track selection codes

The data read from the card is returned in the buffer *buf*. On return, *buf->len* gives the actual length of the string read. Refer to ISO Standard 7813 for a description of the data formats. This function returns when either track data is available or the time-out set by the appropriate `DEV_ioctl` function is reached.
The operation may return tracks read since the last execution of this token. If the VM has this track buffering capability, only one swipe of the card shall be buffered and all the track buffers will be cleared after their contents have been returned by this function (even if all tracks were not requested). The format returned by `MAG_read` for each track is a track number (1,2 or 3), a length byte and the data whose size is specified by the length byte. If multiple tracks are requested, there are multiple instances of the above structure concatenated in the order they were requested. The data is returned in ASCII format with STX/ETX and LRC delimiters removed. Non-bcd digits are left unconverted.

For example :

```
BYTES FROM CARD          BYTES DELIVERED TO APPLICATION
B1 23 4D 7E 5F xx   =>   31 32 33 34 0D 37 0E 35
^            ^ ^LRC
STX          ETX
```

Ior -21 (`XCP_UNSUPPORTED_OPERATION`) shall be returned if the reader does not support one or more of the requested tracks and the content of the returned buffer shall be undefined. If the requested tracks are supported by the reader but are not present on the card swiped, then

ior 0 is returned and the buffer contains those tracks that are available on the card.
If an error occurs while reading one or more of the requested tracks supported by both reader and card swiped, ior -31215 (`XCP_MAG_COM_ERROR`) is returned and the length field of the corresponding tracks in the returned buffer is set to zero. In this case the data of the tracks that were read successfully is available in the returned buffer.

int MAG_read (int *tracks***, BUF_Handle** *buf***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*tracks*	track specification
output	*buf*	reception buffer

Example

See MAG_write.

3.7.2 MAG_write

Write one ISO magstripe. The data is in the buffer *buf* and shall be written to *track* (1-3) when the user swipes the magstripe. If no card is swiped within the time-out period set by the appropriate `DEV_ioctl` function, ior -32766 (`XCP_DEV_TIMEOUT`) is returned.
The data is written in ASCII format with STX/ETX and LRC delimiters removed. Non-bcd digits are left unconverted.

For example :

```
BYTES DELIVERED BY APPLICATION      BYTES TO CARD
31 32 33 34 0D 37 0E 35          => B1 23 4D 7E 5F xx
                                    ^           ^ ^LRC
                                    STX         ETX
```

int MAG_write (int *track***, BUF_Handle** *buf***);**

Return Value

0 if successful, otherwise a device dependent ior (see Appendix B)

Parameters

input	*track*	track specification
output	*buf*	input buffer

Example

```
#include <epic.h>
#include <dbg.h>
#include <mag.h>
#include <icc.h>

void app(void)
{
   BUF_Object buf_data ={0,256,0};
   char str_data[256];
```

```
    unsigned int i ;
    buf_data.data = str_data ;

    // Opening the devices
    DEV_open(DEV_DEBUG);
    DEV_open(DEV_MAGSTRIPE);

    // copying a card track
    DBG_CR ; DBG_STR("Insert card to copy"); DBG_CR;
    for(i=0;MAG_read(2,&buf_data)!=0 && i<500; i++);
    DBG_STR(" i= ");  DBG_INT(i); DBG_CR;
    DBG_STR(" track2 contains :"); DBG_DUMP(&buf_data); DBG_CR;

    DBG_STR("trying to copy track 2, insert new card: "); DBG_CR ;
    for(i=0;MAG_write(2,&buf_data)!=0 && i<500; i++);

    DBG_STR("Testing copy has been done "); DBG_CR ;
    DBG_CR ; DBG_STR("Insert new card"); DBG_CR;
    buf_data.len = 0 ;
    for(i=0;MAG_read(2,&buf_data)!=0 && i<500; i++);
    DBG_STR(" i= ");  DBG_INT(i); DBG_CR;
    DBG_STR(" track2 contains :"); DBG_DUMP(&buf_data); DBG_CR;

    // Closing the devices
    DEV_close(DEV_DEBUG);
    DEV_close(DEV_MAGSTRIPE);

}
```

3.8 Extensible Memory functions

	C function	Main Token invoked
3.8.1	MEM_extend	EXTEND
3.8.2	MEM_extend_bytes	CEXTEND
3.8.3	MEM_move	MOVE
3.8.4	MEM_release	RELEASE

Table 11: MEM functions

3.8.1 MEM_extend

Extend the ‘rubber band’ memory buffer by *len* cells and return the address of the first byte in the allocated memory. The returned pointer is aligned correctly for all data types (cell aligned).

void *MEM_extend (int *len***);**

Return Value

address of the first byte of the allocated memory

Parameters

input	*len*	number of cells to allocate

Exceptions

XCP_OUT_OF_MEMORY	insufficient memory available

Example

```
#include <mem.h>

char *str_dyn;
str_dyn = (char*) MEM_extend(10);
```

3.8.2 MEM_extend_bytes

Extend the 'rubber band' memory buffer by *len* bytes, and returns the address of the first byte in the allocated memory. The returned pointer is not necessary aligned as would be the case with `MEM_extend`.

void *MEM_extend_bytes (int *len***);**

Return Value

Address of the first byte of the allocated memory

Parameters

input	*len*	number of bytes to allocate

Exceptions

XCP_OUT_OF_MEMORY	insufficient memory available

Example

```
#include <mem.h>

char *str_dyn;
str_dyn = (char*) MEM_extend_bytes(10);
```

3.8.3 MEM_move

If *len* is greater than zero, copy *len* bytes from *src* to *dst*, non-destructively if the areas overlap.

void * MEM_move (void * *dst*, **void *** *src*, **int** *len*);

Return Value

Pointer to the destination.

Parameters

input	*dst*	addr where the byte stream must be copied
	src	addr of the byte stream to copy
	len	len of the byte stream to copy

Remark

The destination and origin string may overlap.

Example

```
#include <mem.h>

char str[10];
MEM_move(str, "Hello", 5);
```

3.8.4 MEM_release

Release the memory allocated through `MEM_extend` or `MEM_extend_bytes`. *addr* is the pointer to the address of the byte stream to release and is set to `NULL`.

void MEM_release (void ******addr*);**

Return Value

none

Parameters

input	*addr*	pointer to the address of the byte stream to release

Example

```
#include <mem.h>

char *str_dyn;
str_dyn = (char *) MEM_extend(10);
MEM_release(&str_dyn);
```

3.9 Module Handling Services

	C function	Main token used
3.9.1	MOD_arg_execute	modexecute
3.9.2	MOD_avail	dbavail
3.9.3	MOD_begin	modinitialise
3.9.4	MOD_card_execute	modcardexecute
3.9.5	MOD_changed	modchanged
3.9.6	MOD_data	modappend
3.9.7	MOD_delete	moddelete

Table 12: Module functions

	C function	Main token used
3.9.8	MOD_execute	modexecute
3.9.9	MOD_get_aid	dbstrfetch
3.9.10	MOD_get_flag	dbcfetchlit
3.9.11	MOD_get_version	dbstrfetch
3.9.12	MOD_register	modregister
3.9.13	MOD_release	modrelease

Table 12: Module functions

3.9.1 MOD_arg_execute

Load and execute a module from the module repository and pass a number of arguments to the entry point of this module. If the entry point of the module has a return value, it can be retrieved in the global variable `MOD_return`.

int MOD_arg_execute(BUF_Handle *aid***, int** *n_args***,...);**

Return Value

-1 if the module was loaded successfully, otherwise 0

Parameters

input	*aid*	module ID
	n_args	number of arguments passed
	...	arguments

Example

```
// The following example uses MOD_arg_execute to call a function which is
// the entry point of a module having Module ID F208000014
#include <dbg.h>
#include <mod.h>

void app(void)
{
   char str_aid[17] ;
   BUF_Object buf_aid = {0,17,0} ;
   int ret_val ;

   buf_aid.data= str_aid;
   buf_aid.len= 5 ;

   str_aid[0]= 0xF2;
   str_aid[1]= 0x08;
   str_aid[2]= 0x00;
   str_aid[3]= 0x00;
   str_aid[4]= 0x14;

   // Open debug device
   DEV_open(DEV_DEBUG);

   // four arguments
   ret_val= MOD_arg_execute(&buf_aid,4,'i',8,9,10);
```

```
    DBG_STR("Execution flag must be -1: "); DBG_INT(ret_val);DBG_CR;
    DBG_STR("Return value must be 24: "); DBG_INT(MOD_return);DBG_CR;

    // three arguments
    ret_val= MOD_arg_execute(&buf_aid,3,'b',11,&buf_aid);
    DBG_STR("Execution flag must be -1: "); DBG_INT(ret_val);DBG_CR;
    DBG_STR("Return value must be 16: "); DBG_INT(MOD_return);DBG_CR;

    // close debug device
    DEV_close(DEV_DEBUG);
}
```

3.9.2 MOD_avail

Return the number of modules currently available in the module database.

unsigned int MOD_avail(void);

Return Value

number of modules in repository

Example

```
// This example shows how to implement a function that dumps information
// about all the modules contained in the repository.
void dump_mdf(void)
{
   int i;
   BUF_Object buf_temp= {0,16,0};
   char str_temp[16];
   buf_temp.data= str_temp;
   // Open debug device
   DEV_open(DEV_DEBUG);
   DBG_STR("Modules in repository:");DBG_CR;
   for (i=0; i< MOD_avail(); i++ ) {
      MOD_get_aid(i,&buf_temp);
      DBG_STR("Aid: "); DBG_DUMP(&buf_temp);DBG_CR;
      DBG_STR("Flag:"); DBG_HEX(MOD_get_flag(i));DBG_CR;
      DBG_STR("Version:"); DBG_HEX(MOD_get_version(i));DBG_CR;
   }
   DEV_close(DEV_DEBUG);
}
```

3.9.3 MOD_begin

Prepare the OTA Virtual Machine for acquisition of a new module.

void MOD_begin (void);

Return Value

none

Parameters

none

3.9.4 MOD_card_execute

Load and execute the module at *module_addr. module_addr* is the address of an OTA module delivered from the card into internal storage. If the module executed has a return value, it can be retrieved from the global variable `MOD_return`.

int MOD_card_execute(unsigned char* *module_addr*, **int** *n_args*,**...);**

Return Value

-1 if the module was loaded successfully, otherwise 0

Parameters

input	*module_addr*	address of the module in internal storage
	n_args	number of arguments passed to the module
	...	arguments

Example

```
// In the following example, module_addr points to a module of which
// the entry point is implemented as follows :
int app(int arg1, int arg2) {
   return arg1+arg2;
}

// The function call is the following :
if ((MOD_card_execute(module_addr,2,4,2)==-1) && (MOD_return==6)){
   DBG_STR("OK");
}
```

3.9.5 MOD_changed

Return a value which indicates whether modules have been registered since the last execution of `MOD_changed` (see OTA. Volume 1, Section 6.1). Bits zero through seven of the value returned define which module classes have been registered in the module repository since the last execution of this function. For example, a module registered with an initial AID byte which is F4 will set bit four in the return value. Bits 8 through 31 are reserved for future expansion.

unsigned MOD_changed (void);

Return Value

value indicating whether any classes of modules have been updated.

Parameters

none

3.9.6 MOD_data

Append *buf.len* bytes pointed to by *buf.data* at the end of the module acquisition buffer. Exception XCP_OUT_OF_MEMORY will be thrown if the data can not be appended.

void MOD_data (BUF_Handle *buf*);

Parameters

input	*buf*	buffer containing the data to be appended

Exceptions

XCP_OUT_OF_MEMORY	the acquisition buffer capacity is exceeded

Example

```
// In the next example we assume that the address 'str_load9_1_mdf' points
// to a string representing a hexadecimal data coding an mdf file.
#include <mod.h>
void app(void)
{
   BUF_Object buf_module = {214,214,0} ;
   MOD_begin();
   buf_module.data= (char*)str_load9_1_mdf ;
   MOD_data(&buf_module);
   MOD_register();
}
```

3.9.7 MOD_delete

Delete the module whose AID is specified by *aid* from the module repository.

void MOD_delete (BUF_Handle *aid***);**

Return Value:

none

Parameters

input	*aid*	buffer containing the aid of the module to delete

Exceptions

XCP_MOD_OPEN_ERROR	cannot open repository
XCP_MOD_DELETE_NOT_ALLOWED	deletion of module not allowed
XCP_MOD_NOT_AVAILABLE	module not in repository

Example

```
#include <mod.h>
void app(void)
{
   BUF_Object buf_aid = {5,5,0} ;
   char str_aid[5];
   int xcp;
   buf_aid.data= str_aid;
   str_aid[0]= 0xF2;   str_aid[1]= 0x01;
   str_aid[2]= 0x00;   str_aid[3]= 0x01;
   str_aid[4]= 0x01;
   // Open debug device
   DEV_open(DEV_DEBUG);
   if ((xcp = XCP_arg_catch(1,(FUNC)MOD_delete,&buf_aid)) == 0){
      DBG_STR("Module deleted");
```

```
    }
    else {
       DBG_STR("Module could not be deleted : "); DBG_INT(xcp);
    };
    DBG_CR;
    DEV_close(DEV_DEBUG);
}
```

3.9.8 MOD_execute

Load and execute a module from the module repository using the AID specified by *aid*. The entry point of this module must not have parameters. If it has a return value, it can be retrieved in the global variable `MOD_return`.

int MOD_execute (BUF_Handle *aid***);**

Return Value:

-1 if the module was loaded successfully, otherwise 0

Parameters

input	*aid*	module ID

Example

```
#include <dbg.h>
#include <mod.h>

void app(void)
{
   char str_aid[17] ;
   BUF_Object buf_aid = {0,17,0} ;
   buf_aid.data= str_aid;
   buf_aid.len= 5 ;
   str_aid[0]= 0xF2;
   str_aid[1]= 0x00;
   str_aid[2]= 0x00;
   str_aid[3]= 0x00;
   str_aid[4]= 0x14;
   ret_val= MOD_execute(&buf_aid);
}
```

3.9.9 MOD_get_aid

Given the module record number *rec_nr* in the module database, `MOD_get_aid` retrieves the aid value and stores it in the user's allocated `BUF_Handle`. The buffer length is automatically updated (no trailing spaces).

void MOD_get_aid(unsigned int *rec_nr***, BUF_Handle** *buf_aid***);**

Parameters

input	*rec_nr*	*record number*
output	*buf_aid*	*buffer to receive the aid*

Exceptions

`XCP_DB_INVALID_RECORD`	the record number is outside the available range

Example

See MOD_avail.

3.9.10 MOD_get_flag

Given the module record number *rec_nr* in the module database, `MOD_get_flag` returns the flag value.

char MOD_get_flag(unsigned int *rec_nr***);**

Return Value:

flag of the module

Parameters

input	*rec_nr*	module record number

Exceptions

`XCP_DB_INVALID_RECORD`	the record number is outside the available range

Example

See MOD_avail.

3.9.11 MOD_get_version

Given the module record number *rec_nr* in the module database, `MOD_get_version` returns the version number.

int MOD_get_version(unsigned int *rec_nr***);**

Return Value:

version of the module

Parameters

input	*rec_nr*	module record number

Exceptions

`XCP_DB_INVALID_RECORD`	the record number is outside the available range

Example

See MOD_avail.

3.9.12 MOD_register

Register the module acquisition buffer in the module repository.

void MOD_register (void);

Return Value:

none

Parameters

none

Exceptions

XCP_MOD_CANNOT_ADD	cannot add module
XCP_MOD_OPEN_ERROR	cannot open repository
XCP_MOD_INVALID_LENGTH	invalid module length
XCP_MOD_INVALID_MDF	not a valid module
XCP_MOD_INVALID_TOKEN	invalid token

Example

See MOD_data.

3.9.13 MOD_release

Release the resources used by the internal module buffer. This is required if premature loading of a module must be terminated by the application without registering the module in the module repository.

void MOD_release (void);

Return Value:

none

Parameters

none

3.10 Message services

	C function	Main Token invoked
3.10.1	MSG_avail	DBAVAIL
3.10.2	MSG_choose_lang	CHOOSELANG
3.10.3	MSG_code_page	CODEPAGE
3.10.4	MSG_delete	MSGDELETE
3.10.5	MSG_get	MSGFETCH
3.10.6	MSG_get_ascii_name	DBSTRFETCH
3.10.7	MSG_get_code_name	DBSTRFETCH
3.10.8	MSG_get_code_page	DBCFETCHLIT

Table 13: MSG functions

	C function	Main Token invoked
3.10.9	MSG_get_symbol	DBSTRFETCH
3.10.10	MSG_init	MSGINIT
3.10.11	MSG_load	MSGLOAD
3.10.12	MSG_load_page	LOADPAGE
3.10.13	MSG_update	MSGUPDATE

Table 13: MSG functions

3.10.1 MSG_avail

Returns the total number of languages available in the language database.

unsigned int MSG_avail();

Return Value

number of languages available

Example

```
#include <msg.h>
#include <dbg.h>

void app(void)
{
   DEV_open(DEV_DEBUG);
   DBG_STR("Must reflect the total number of languages already loaded: ");
   DBG_INT(MSG_avail()); DBG_CR;
   DEV_close(DEV_DEBUG);
}
```

3.10.2 MSG_choose_lang

Select the language whose ISO 639 language code is given by the two characters at *str_lang.* If the function returns -1, the language was found and is now the current language. Otherwise, the calling program should select another language. At least one language (the terminal's native language) shall always be available. It is the responsibility of the program using `MSG_choose_lang` to make the current message selection available to any client program using `MSG_update`.

int MSG_choose_lang (char* *str_lang***);**

Return Value

-1 if the language was found, otherwise 0

Parameters

input	*str_lang*	ISO 639 language code in a 2-character string

Example

```
#include <msg.h>
#include <dbg.h>

// choose the french language
```

```
int flag;
DEV_open(DEV_DEBUG);
flag = MSG_choose_lang("fr");
DBG_STR("Flag : "); DBG_INT(flag); DBG_CR;
DEV_close(DEV_DEBUG);
```

3.10.3 MSG_code_page

Select the resident code page *codep*. Code pages are numbered according to ISO 8859 (0 = common character set, 1 = Latin 1 etc.).

int MSG_code_page (unsigned int *codep***);**

Return Value

-1 if the code page has been selected

Parameters

input	*codep*	code page number according to ISO 8859

3.10.4 MSG_delete

Delete the language database entry and resident messages for the language whose ISO 639 language code is given by the two bytes pointed by *language*.

int MSG_delete (char* *language***);**

Return Value

-1 if the language was found and deleted successfully, 0 if the language was not found

Parameters

input	*buf_table*	buffer containing the ISO 639 language code

Example

```
#include <msg.h>
#include <dbg.h>

// Removing the english message table from the database
MSG_delete("en");
// trying to choose the english language
flag = MSG_choose_lang("en");
DBG_STR("Flag returned by MSG_choose_lang = ");
DBG_INT(flag);
```

3.10.5 MSG_get

Return the string parameter for message *num.* The message is retrieved from the transient buffer, if present. If not, the message is retrieved from the currently selected language table. Trailing spaces are removed from the length of the string. The data field of *buf_msg* has to be allocated by the calling function. If no message is found, the returned buffer will be empty.

void MSG_get (unsigned int *num***, BUF_Handle** *buf_msg***);**

Return Value

none

Parameters

input	*num*	message number
output	*buf_msg*	buffer containing the message string

Remark

If message *num* has not been provided in the currently selected language, the function returns string parameters for a zero-length message.

Example

For an example of `MSG_get` refer to the example at the end of this section.

3.10.6 MSG_get_ascii_name

Given the record number in the language database, `MSG_get_ascii_name` stores in *buf* the ASCII string parameters for the name (in 7 bit ASCII codes) of the referenced language. Trailing spaces are removed, maximum length is 16 bytes.

void MSG_get_ascii_name (unsigned int *rec_nr***, BUF_Handle** *buf***);**

Parameters

input	*rec_nr*	language record number.
output	*buf*	buffer to receive the name

Exceptions

`XCP_DB_INVALID_RECORD`	the record number is outside the available range

Example

```
#include <buf.h>
#include <msg.h>
#include <dbg.h>

void app(void)
{
   BUF_Object buf_name = {0,33,0};
   char str_name [33];
   buf_msg.data = str_name ;
   DEV_open(DEV_DEBUG);
   MSG_get_ascii_name(0, &buf_msg);
   DBG_STR("Ascii name representing first language in database:");
   DBG_DUMP_ASCII(&buf_msg);DBG_CR;
   DEV_close(DEV_DEBUG);
}
```

3.10.7 MSG_get_code_name

Given the record number in the language database, `MSG_get_code_name` stores in *buf* the ASCII string parameters for the name (in code page characters) of the referenced language**.**

Trailing spaces are removed, maximum length will be 16 bytes.

void MSG_get_code_name (unsigned int *rec_nr***, BUF_Handle** *buf***);**

Parameters

input	*rec_nr*	language record number.
output	*buf*	buffer to store the name

Exceptions

XCP_DB_INVALID_RECORD	the record number is outside the available range

Example

```
#include <buf.h>
#include <msg.h>
#include <dbg.h>

void app(void)
{
   BUF_Object buf_name = {0,33,0};
   char str_name [33];
   buf_msg.data = str_name;
   DEV_open(DEV_DEBUG);
   MSG_get_code_name(0, &buf_msg);
   DBG_STR("Code name representing first language in database:");
   DBG_DUMP_ASCII(&buf_msg);DBG_CR;
   DEV_close(DEV_DEBUG);
}
```

3.10.8 MSG_get_code_page

Returns the ISO 8859 code page number used by the record *rec_nr* in the language database.

char MSG_get_code_page (unsigned int *rec_nr***);**

Parameters

input	*rec_nr*	language record number.

Exceptions

XCP_DB_INVALID_RECORD	the record number is outside the available range

Example

```
#include <msg.h>
#include <dbg.h>

void app(void)
{
   char temp_char ;
   temp_char= MSG_get_code_page(1);
   DEV_open(DEV_DEBUG);
   DBG_STR("Code page of the second record in language DB is:");
   DBG_INT((int)temp_char);DBG_CR;
   DEV_close(DEV_DEBUG);
```

```
}
```

3.10.9 MSG_get_symbol

Store at address *str_lang* the two character symbols (according to ISO 639) referring to the language *rec_nr* in the language database.

void MSG_get_symbol (unsigned int *rec_nr***,char*** *str_lang***);**

Parameters

input	*rec_nr*	language record number.
	str_lang	location to put the two character symbols

Exceptions

XCP_DB_INVALID_RECORD	the record number is outside the available range

Example

```
#include <msg.h>
#include <dbg.h>

void app(void)
{
   char lang_symbol[2];
   DEV_open(DEV_DEBUG);
   MSG_get_symbol(1,lang_symbol);
   DBG_STR("Language symbol of the second record: ");
   DBG_HEX(lang_symbol[0]);DBG_HEX(lang_symbol[1]);DBG_CR;
   DEV_close(DEV_DEBUG);
}
```

3.10.10 MSG_init

Erase the transient messages, numbered from 0x40 to 0xFF from the transient buffer.

void MSG_init (void);

Return Value

none

Parameters

none

Example

```
#include <msg.h>

// Erase all messages in the transient buffer
MSG_init();
```

3.10.11 MSG_load

Install a message table pointed by *str_table*, including message numbers in the range 0x40-0xFF into the transient messages buffer, over-writing any previous messages with the same

message numbers. Messages 0x40-0xFF may be provided only by a terminal application; messages 0xC0-0xFF may be provided by an ICC. The message table must be formatted as described in Section 4.4 of OTA Volume 1.

int MSG_load (char* *str_table***);**

Return Value

-1 if the messages were loaded successfully, 0 if the message table's code page is currently not selected or if the messages are outside the range 0x40-0xFF

Parameters

input	*str_table*	

Prerequisite

The current message selection has been made available through `MSG_choose_lang`. The appropriate code page has been selected using `MSG_code_page`.

Example

For an example of `MSG_load` refer to the example at the end of this section.

3.10.12 MSG_load_page

Install the code page *str_codep* in the terminal.

int MSG_load_page (char* *str_codep***);**

Return Value

-1 if the code page is successfully installed, otherwise 0

Parameters

input	*str_codep*	code page address

3.10.13 MSG_update

Install the message table pointed by *str_table* including message numbers in the range 1-0xFF into the resident language database. If a language with the same code is already present, new messages replace previous messages with the same number.

void MSG_update (char ***str_table*);**

Return Value

none

Parameters

input	*str_table*	address of message table definition, formatted as described in OTA Volume 1, Section 4.4

Exceptions

XCP_TOO_MANY_LANGUAGES	there is not sufficient space for the new language

Example

For an example of the use of `MSG_update` refer to the example at the end of this section.

3.10.14 Message example

The following example explains how to use the different message services:

```
#include <dbg.h>
#include <buf.h>
#include <mem.h>
#include <msg.h>
#include <str.h>
#include <dev.h>

// typedef of structure to hold message table
typedef struct msg_c  {
   unsigned char number;
   char* content;
} msg_content;

// message table containing 5 messages
msg_content english_msg[] = {
   {1,  "Amount:" },
   {2,  "OK?" },
   {3,  "Approved" },
   {40, "Timed Out" },
   {41, "Thank You" },
   {0, 0 }
} ;

void app (void)
{
   BUF_Object buf ={0, 1024, 0};
   BUF_Object msg = {0, 16, 0 };
   char str_msg_table[1024];
   char str_msg[16];
   buf.data = str_msg_table;
   msg.data = str_msg;

   // Open devices
   DEV_open(DEV_DEBUG);
   DEV_open(DEV_DISPLAY);

   // First convert the message table to the format specified
   // in OTA Volume 1.

   // initialise table with english header
   BUF_set_str(&buf,"en");         // ISO 639 code for english
   buf.data[buf.len++] = 0x01 ;    // code page
   buf.data[buf.len++] = 0x07 ;    // length of code name
   BUF_append_str(&buf,"English"); // code name
   buf.data[buf.len++] = 0x07;     // length of ASCII name
   BUF_append_str(&buf,"English"); // ASCII name
```

```
    // add the messages
    k = 0;
    while (english_msg[k].number != 0) {
       buf.data[buf.len++] = english_msg[k].number;   // message number
       buf.data[buf.len++] = (char)(STR_length(english_msg[k].content));
                                                      // message length
       BUF_append_str(&buf,english_msg[k++].content); // message content
    }
    // end of message table
    buf.data[buf.len++] = 0x00;     // message number
    buf.data[buf.len++] = 0x01;     // message length
    buf.data[buf.len++] = 0x00;     // message contents

    // Install the english message table in the message database
    MSG_update(buf.data);

    // Choose english as current language
    MSG_choose_lang("en");

    // Get message 40 and display it on the terminal display
    MSG_get(40,&msg);
    DEV_locate(DEV_DISPLAY,0,0);
    DEV_centred_type(DEV_DISPLAY,&msg,MSG_size());

    // Overload message 40 with a new message
    // We first create a new message table with only message 40
    BUF_set_str(&buf,"en");         // ISO 639 code for english
    buf.data[buf.len++] = 0x01 ;    // code page
    buf.data[buf.len++] = 0x07 ;    // length of code name
    BUF_append_str(&buf,"English"); // code name
    buf.data[buf.len++] = 0x07;     // length of ASCII name
    BUF_append_str(&buf,"English"); // ASCII name
    // add new message 40
    buf.data[buf.len++] = 40;       // message number
    buf.data[buf.len++] = 5;
    BUF_append_str(&buf,"Hello");
    // end of message table
    buf.data[buf.len++] = 0x00;     // message number
    buf.data[buf.len++] = 0x01;     // message length
    buf.data[buf.len++] = 0x00;     // message contents

    // Overload message 40
    flag = MSG_load(buf.data);
    DBG_INT(flag);

    // Get the msg 40 and display it again on the terminal display
    // This time "hello" should appear on the display.
    MSG_get(40,&msg);
    DEV_locate(DEV_DISPLAY,0,0);
    DEV_centred_type(DEV_DISPLAY,&msg,MSG_size());

    // Close devices
    DEV_close(DEV_DEBUG);
    DEV_close(DEV_DISPLAY);

}
```

3.11 Operating system interface functions

	C function	Main Token invoked
3.11.1	OSS_call	OSCALL
3.11.2	OSS_set_callback	SETCALLBACK

Table 14: O.S. interface functions

3.11.1 OSS_call

Call an operating system function *fn* with *len* cell-sized arguments in the array *args*. Individual functions are terminal dependent and are defined in Appendix C. Exception -21 (`XCP_UNSUPPORTED_OPERATION`) will be thrown if a particular function is not supported by the Virtual Machine.

void OSS_call(int *fn***, int** *len***, int** *args[]***);**

Return Value

none

Parameters

input	*fn*	number of the operating system function
	len	length of the input array
	args	address of the input array

Exceptions

`XCP_UNSUPPORTED_OPERATION`	function not supported

3.11.2 OSS_set_callback

Anounce xp as an OTA routine that may be called by the underlying operating system.

void OSS_set_callback(void (**xp***)(void));**

Return Value

none

Parameters

input	*xp*	execution pointer to the routine

3.12 Cryptographic services

	C function	Main Token invoked
3.12.1	SEC_crc	CRYPTO
3.12.2	SEC_des_decryption	CRYPTO
3.12.3	SEC_des_encryption	CRYPTO
3.12.4	SEC_des_key_schedule	CRYPTO
3.12.5	SEC_incr_sha_init	CRYPTO
3.12.6	SEC_incr_sha_terminate	CRYPTO
3.12.7	SEC_incr_sha_update	CRYPTO
3.12.8	SEC_long_shift	CRYPTO
3.12.9	SEC_long_sub	CRYPTO
3.12.10	SEC_mod_exp	CRYPTO
3.12.11	SEC_mod_mult	CRYPTO
3.12.12	SEC_sha	CRYPTO

Table 15: SEC function

3.12.1 SEC_crc

This function calculates the cyclic redundancy check (CRC) over the input data represented by the input buffer *buf_input*. *shift_order* defines the order in which the bits of data bytes are shifted: zero is least significant bit first and non-zero is most significant bit first. *crc_input* is the input for the CRC algorithm, which allows chained computation over multiple blocks and also for preconditioning data for CRC-32. *polynomial* is the polynomial, where the bit representing the most significant term has been discarded (e.g.: the polynomial $x^{16} + x^{12} + x^5 + 1$ is represented by the binary value 1000000100001).

unsigned int SEC_crc (unsigned int *polynomial***, unsigned int** *crc_input***, int** *shift_order***,**
BUF_Handle *buf_input***);**

Return Value

32 bit value of the new CRC computed over the input data

Parameters

input	*polynomial*	polynomial
	crc_input	input for the CRC algorithm
	shift_order	order in which the bits of data bytes are shifted.
	buf_input	input data

3.12.2 SEC_des_decryption

This function performs the DES decryption algorithm. *buf_input* contains the 8 byte input buffer and *buf_key_schedule* contains the 128 byte key schedule buffer computed with the `SEC_des_key_schedule` function.

void SEC_des_decryption (BUF_Handle *buf_result***, BUF_Handle** *buf_input***,**
BUF_Handle *buf_key_schedule***);**

Return Value

none

Parameters

input	*buf_input*	input buffer
	buf_key_schedule	key schedule buffer
output	*buf_result*	result

3.12.3 SEC_des_encryption

This function performs the DES encryption algorithm. *buf_input* contains the 8 byte input buffer and *buf_key_schedule* contains the 128 byte key schedule buffer computed with the `SEC_des_key_schedule` function.

void SEC_des_encryption (BUF_Handle *buf_result***, BUF_Handle** *buf_input***,**
BUF_Handle *buf_key_schedule***);**

Return Value

none.

Parameters

input	*buf_input*	input buffer
	buf_key_schedule	key schedule buffer
output	*buf_result*	result

3.12.4 SEC_des_key_schedule

The function performs the DES key schedule algorithm. *key* is the 8 byte key buffer including parity bits and *key_schedule* is the 128 byte key schedule output buffer.

void SEC_des_key_schedule (BUF_Handle *key_schedule***, BUF_Handle** *key***);**

Return Value

none

Parameters

input	*key*	8 byte key buffer
output	*key_schedule*	128 byte key schedule output buffer

3.12.5 SEC_incr_sha_init

The incremental SHA-1 calculation is devided in three steps. `SEC_incr_sha_init` performs the initialisation step. It initialises the *init_hash* buffer with the five 32 bit offsets required by this algorithm.

void SEC_incr_sha_init (BUF_Handle *init_hash***);**

Return Value

none

Parameters

output	*init_hash*	initialisation buffer

3.12.6 SEC_incr_sha_terminate

The incremental SHA-1 calculation is devided in 3 steps. `SEC_incr_sha_terminate` performs the termination step. For this step the *remain_input* contains the remaining bytes of the complete input buffer. The length does not have to be a mutliple of 64. The output buffer *hash* must not overlap the input buffer *remain_input*.

void SEC_incr_sha_terminate (BUF_Handle *hash***, BUF_Handle** *remain_input***);**

Return Value

none

Parameters

input	*remain_input*	remaining input bytes
output	*hash*	hash result

3.12.7 SEC_incr_sha_update

The incremental SHA-1 calculation is devided in 3 steps. `SEC_incr_sha_update` performs the update step. The update step may be repeated as often as necessary. The length of the *input* buffer has to be a multiple of 64 bytes. The *intermediate_hash* buffer must not overlap the input buffer and is passed between the different steps.

void SEC_incr_sha_update (BUF_Handle *intermediate_hash***, BUF_Handle** *input***);**

Return Value

none

Parameters

input	*key*	8 byte key buffer
output	*key_schedule*	128 byte key schedule output buffer

3.12.8 SEC_long_shift

This function shifts the binary string *buf_shift* by *n* bits. If *n* is positive the string is shifted *n* places to the left, and the least significant bits are filled with zero. If *n* is negative, the string is shifted *n* places to the right, propagating the most significant bit.

void SEC_long_shift (BUF_Handle *buf_shift***, int** *n***);**

Return Value

none

Parameters

input	*buf_shift*	binary data to be shifted
	n	value used for shift

output	*buf_result*	result of the algorithm

Remarks:

The input value is any multiple of 8 bits, up to and including 1024 bits. The value must be in big endian byte order.

Example:

```
#include <sec.h>
#include <dbg.h>
#include <buf.h>

void app(void)
{
   BUF_Object buf_result= {0,20,0};
   char data_result[20];
   buf_result.data= data_result;

   BUF_set_counted_str(&buf_result,"\x00\x00\x00\x00\x00\x00\x04\x00",8);
   SEC_long_shift(&buf_result,-3);
}
```

3.12.9 SEC_long_sub

This function subtracts two numbers. The value represented by *buf_subtract* is subtracted from the value represented by *buf_input*, and the result is stored in *buf_subtract.*

void SEC_long_sub(BUF_Handle *buf_subtract***,BUF_Handle** *buf_input***);**

Return Value

none

Parameters

input	buf_substract	value to substract
	buf_input	input value
output	*buf_substract*	result of the algorithm.(subtract = input-subtract)

Remarks:

The input values are represented by byte strings, and may be any multiple of 8 bits up to and including 1024 bits. Both input must have the same length. The values must be in big endian byte order.

Example:

```
#include <sec.h>
#include <dbg.h>
#include <buf.h>

void app(void)
{
   BUF_Object buf_result= {0,20,0};
   BUF_Object buf_x = {0,20,0};
   char data_result[20];
   char data_x[20];
```

```
    buf_result.data= data_result;
    buf_x.data= data_x;

    BUF_set_str(&buf_x,"123456");
    BUF_set_str(&buf_result,"654321");
    SEC_long_sub(&buf_result,&buf_x);
 }
```

3.12.10 SEC_mod_exp

This function raises an unsigned value x represented by *buf_x* to a power given by the unsigned exponent *y*, where the product is reduced using the modulus z represented by *buf_z* (result = MOD(x^y,z)).

void SEC_mod_exp (BUF_Handle *buf_result***, BUF_Handle** *buf_x***, unsigned int** *y***,**
BUF_Handle *buf_z***);**

Return Value

none

Parameters

input	*buf_result*	location to put the result
	buf_x	x value
	y	exponent
	buf_z	z value
output	*buf_result*	result

Remarks:

The input value *buf_x* and modulus *buf_z* contain byte strings that may be any multiple of 8 bits up to and including 1024 bits. The values must be in big endian byte order. The buffer *buf_result* may not overlap with any of the input buffers.

Example:

```
#include <sec.h>
#include <dbg.h>
#include <buf.h>

void app(void)
{
   BUF_Object buf_x = {0,20,0};
   BUF_Object buf_z = {0,20,0};
   BUF_Object buf_result= {0,20,0};
   char data_x[20];
   char data_z[20];
   char data_result[20];

   buf_x.data= data_x;
   buf_z.data= data_z;
   buf_result.data= data_result;

   BUF_set_str(&buf_x,"123456789");
   BUF_set_str(&buf_z,"456456456");
   SEC_mod_exp(&buf_result, &buf_x, 4, &buf_z);
```

```
}
```

3.12.11 SEC_mod_mult

This function performs a multiplication of two unsigned values represented by the buffers *buf_x* and *buf_y*, where the product is reduced using the modulus z represented by the buffer *buf_z* (result = MOD(x*y,z)).

void SEC_mod_mult(BUF_Handle *buf_result***, BUF_Handle** *buf_x***, BUF_Handle** *buf_y***, BUF_Handle** *buf_z***);**

Return Value

none

Parameters

input	*buf_result*	result buffer
	buf_x	x value
	buf_y	y value
	buf_z	z value
output	*buf_result*	result buffer

Remarks:

The input values (*buf_x*, *buf_y*, *buf_z*) are all the same length. Each string can be any multiple of 8 bits up to and including 1024 bits. The values must be in big endian byte order. *buf_x*, *buf_y*, *buf_z* may be the same string, but *buf_result* may not overlap with any of the input buffers.

Example:

```
#include <sec.h>
#include <dbg.h>
#include <buf.h>

void app(void)
{
   BUF_Object buf_x = {0,20,0};
   BUF_Object buf_y = {0,20,0};
   BUF_Object buf_z = {0,20,0};
   BUF_Object buf_result= {0,20,0};
   char data_x[20];
   char data_y[20];
   char data_z[20];
   char data_result[20];

   buf_x.data= data_x;
   buf_y.data= data_y;
   buf_z.data= data_z;
   buf_result.data= data_result;

   // SEC_mod_mult
   BUF_set_str(&buf_x,"\x04\x05");
   BUF_set_str(&buf_y,"\x05\x04");
   BUF_set_str(&buf_z,"\x07\x08");
```

```
    SEC_mod_mult(&buf_result,&buf_x,&buf_y,&buf_z);
}
```

3.12.12 SEC_sha

Performs Secure Hash Algorithm (SHA-1). This algorithm takes as input the message *buf_input* of arbitrary length and produces a 20-byte hash value.

void SEC_sha(BUF_Handle *buf_hash_value*, **BUF_Handle** *buf_input***);**

Return Value

none

Parameters

input	*buf_input*	input value
	buf_hash_value	location to put the computed hash
output	*buf_hash_value*	hash value

Remarks:

The buffer *buf_hash_value* must not overlap the *buf_input* buffer.

Example:

```
#include <sec.h>
#include <dbg.h>
#include <buf.h>

void app(void)
{
   BUF_Object buf_z = {0,20,0};
   BUF_Object buf_result= {0,20,0};
   char data_z[20];
   char data_result[20];

   buf_z.data= data_z;
   buf_result.data= data_result;
   // SEC_sha
   BUF_set_str(&buf_z,"\x07\x05\x33\x33\x33\x33\x33");
   SEC_sha(&buf_result,&buf_z);
}
```

3.13 Socket services

	C function	Main Token invoked
3.13.1	SKT_arg_run	idosocket
3.13.2	SKT_control	plugsocket
3.13.3	SKT_plug	plugsocket
3.13.4	SKT_run	idosocket

Table 16: Socket functions

3.13.1 SKT_arg_run

Perform the action of socket number *num_socket*, where *num_socket* is zero through 63. The socket *num_socket* must mandatory handle an int function (int) or equivalent (one argument and one mandatory return value that must hold in one cell).

int SKT_arg_run (unsigned *num_socket*, **int** *arg* **);**

Return Value

the returned value of the socket executed

Parameters

input	*num*	number of the socket you want to execute
	arg	argument to pass to the procedure

Exceptions

XCP_SOC_INV_SOCKET	the number of the socket is outside the range 0-63

Example: see SKT_plug

3.13.2 SKT_control

Set *fn* to be the procedure to run when attempting to validate whether a socket can be plugged or not. This function must mandatory match the prototype: `int (*fn)(int nr_sckt)`.

int SKT_control (int (*fn) (**int** num_socket)**);**

Return Value

-1 if the function *fn* was plugged successfully, otherwise 0

Parameters

input	*fn*	function to run when attempting to validate whether a socket can be plugged or not. The parameter of the *fn* function is generated automatically.

Example:

```
#include <skt.h>
```

```
#include <dev.h>
#include <dbg.h>
#include <buf.h>

int test_plug(int socket)
{
   DBG_STR(" plug"); DBG_INT(socket); DBG_CR;
   return -1;
}

void sayhello(void)
{
   DEV_type(DEV_DISPLAY,"Hello");
   return;
}

void app(void)
{
   int result;
   DEV_open(DEV_DEBUG);
   DEV_open(DEV_DISPLAY);
   SKT_control(test_plug);
   // From now on each call to the function pluggable will be replaced
   // by a call to test_plug
   // ....
}
```

3.13.3 SKT_plug

Set the handler of socket *num* to be *fn*. Whether the socket is plugged or not, is controlled by the procedure installed by `SKT_control`. The procedure handled by the socket must be a `void function(void)` or an `int function(int)`.

int SKT_plug (unsigned *num***, FUNC** *fn***);**

Return Value

-1 if the function *fn* was plugged successfully, otherwise 0

Parameters

input	*num*	number of the socket between 0 and 63
	fn	Function to be plugged as a socket.

Exceptions

`XCP_SOC_INV_SOCKET`	the number of the socket is outside the range 0-63.

Example:

```
#include <skt.h>
#include <dev.h>
#include <buf.h>

void sayhello(void)
{
   DEV_type(DEV_DISPLAY,"Hello");
   return;
```

```
}
void sayhello2(void)
{
   DEV_type(DEV_DISPLAY,"Hello Again");
   return;
}
int new_socket(int i)
{
    return i+1 ;
}
void app(void)
{
   DEV_open(DEV_DISPLAY);
   DEV_open(DEV_DEBUG);
   if (SKT_plug(1, sayhello == -1) {
      DBG_STR(" Plug OK 1");
   }
   if (SKT_plug(2, sayhello2) == -1) {
      DBG_STR(" Plug OK 2");
   }
   // Execute the socket 2 and the socket 1
   SKT_run(2);
   SKT_run(1);

   if (SKT_plug(1, (FUNC) new_socket)==-1){
      DBG_STR("Must be 3: ");DBG_INT(SKT_arg_run(1,2)); DBG_CR;
      DBG_STR("Must be 4: ");DBG_INT(SKT_arg_run(1,3)); DBG_CR;
      DBG_STR("Must be 5: ");DBG_INT(SKT_arg_run(1,4)); DBG_CR;
   }
   DEV_close(DEV_DISPLAY);
   DEV_close(DEV_DEBUG);
}
```

3.13.4 SKT_run

Execute the action associated with socket *num*. The socket *num* must mandatory handle a `void function(void)`.

void SKT_run (unsigned *num***);**

Return Value

none

Parameters

input	*num*	The number of the socket you want to execute.

Exceptions

`XCP_SOC_INV_SOCKET`	the number of the socket is outside the range 0-63.

Example

see above `SKT_plug`

3.14 String functions

	C function	Main token invoked
3.14.1	STR_append_bintohex	
3.14.2	STR_append_hextoa	
3.14.3	STR_append_hextobin	
3.14.4	STR_append_itoa	NMBR
3.14.5	STR_atoi	TONUMBER
3.14.6	STR_blank	FILL
3.14.7	STR_compare	COMPARE
3.14.8	STR_erase	FILL
3.14.9	STR_fetch_bcd	BCDFETCH
3.14.10	STR_fetch_bn	BNFETCH
3.14.11	STR_fetch_cn	CNFETCH
3.14.12	STR_fill	FILL
3.14.13	STR_itoa	NMBR
3.14.14	STR_length	
3.14.15	STR_minus_trailing	MINUSTRAILING
3.14.16	STR_minus_zeros	MINUSZEROS
3.14.17	STR_scan	SCAN
3.14.18	STR_skip	SKIP
3.14.19	STR_store_bcd	BCDSTORE
3.14.20	STR_store_bn	BNSTORE
3.14.21	STR_store_cn	CNSTORE

Table 17: STR functions

3.14.1 STR_append_bintohex

Convert the binary string represented by *buf_bin* into a corresponding ascii string where each character represents a hexadecimal nibble. Append this converted string at the end of *buf_hex.*

void STR_append_bintohex (BUF_Handle *buf_hex***, BUF_Handle** *buf_bin***);**

Return Value

none

Parameters

input	*buf_hex*	ascii buffer
	buf_bin	binary buffer to be appended
output	*buf_hex*	resulting ascii buffer

Remarks

The length of the ascii buffer must be long enough to receive the result, no check is made.

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>
void app(void)
{
   BUF_Object buf_bin = {2,2,0};
   BUF_Object buf_hex = {0,20,0};
   char str_hex[20];
   buf_hex.data = str_hex;
   DEV_open(DEV_DEBUG);
   BUF_set_str(&buf_hex,"F201");
   buf_bin.data = "\xF1\xE9";
   STR_append_bintohex(&buf_hex,&buf_bin);
   DBG_STR("Must be F201F1E9: ");
   DBG_DUMP_ASCII(&buf_bin);
   DEV_close(DEV_DEBUG);
}
```

3.14.2 STR_append_hextoa

Convert the unsigned integer *num* to an ASCII string representing a hexadecimal number and append it to the buffer *buf_ascii* using *len_append* bytes. If *len_append* is greater than the length required, the number string will be left padded with character '0'. If *len_append* is too short, the number string will be left truncated, and appended to *buf_ascii*.

void STR_append_hextoa (unsigned int *num***, BUF_Handle** *buf_ascii***, char** *len_append***);**

Return Value

none

Parameters

input	*num*	number to append
	buf_ascii	input buffer
	len_append	length of the ascii string to be appended
output	*buf_ascii*	output buffer

Exceptions

XCP_OUT_OF_MEMORY	buffer size exceeded

Example

```
#include <buf.h>
```

```
#include <str.h>
#include <dbg.h>
void app(void)
{
   BUF_Object buf_ascii = { 0, 25, 0 };
   char str_ascii[25];
   buf_ascii.data = str_ascii;
   DEV_open(DEV_DEBUG);
   BUF_set_str(&buf_ascii,"111");
   STR_append_hextoa (1000,&buf_ascii,6);
   DBG_CR;
   DBG_STR("buf_ascii (= ? 1110003E8)= ");
   DBG_DUMP_ASCII(&buf_ascii);
   DEV_close(DEV_DEBUG);
}
```

3.14.3 STR_append_hextobin

Convert the ASCII string represented by *buf_hex* and containing hexadecimal characters (from '0' to 'F') into the corresponding binary string. Append the converted string at the end of *buf_bin*.

void STR_append_hextobin (BUF_Handle *buf_bin***, BUF_Handle** *buf_hex***);**

Return Value

none

Parameters

input	*buf_bin*	binary buffer
	buf_hex	ascii buffer to be converted
output	*buf_bin*	modified binary buffer

Remarks

The binary buffer size must be long enough to receive the result, no check is made. If the hexadecimal ascii string is of odd length, the last digit of the bin buffer will be undefined.

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>
void app(void)
{
   BUF_Object buf_hex = { 0, 20, 0 };
   BUF_Object buf_bin= { 0, 10, 0 };
   char str_hex[20];
   char str_bin[10];
   buf_bin.data = str_bin;
   buf_hex.data = str_hex;
   DEV_open(DEV_DEBUG);
   BUF_set_str(&buf_bin,"\xF2\x01");
   BUF_set_str(&buf_hex,"F1");
   STR_append_hextobin(&buf_bin,&buf_hex);
```

```
    DBG_STR("Must be F2 01 F1: ");
    DBG_DUMP(&buf_bin);
    DEV_close(DEV_DEBUG);
}
```

3.14.4 STR_append_itoa

Convert the unsigned integer *num* to an ASCII string and append it to the buffer *buf_ascii* using *len_append* bytes. If *len_append* is greater than the length required, the number string will be left padded with character '0'. If *len_append* is too short, the number string will be left truncated, and appended to *buf_ascii*.

void STR_append_itoa (unsigned int *num***, BUF_Handle** *buf_ascii***, char** *len_append***);**

Return Value

None

Parameters

input	*num*	number to append
	buf_ascii	input buffer
	len_append	length of the ascii string to be appended
output	*buf_ascii*	output buffer

Exceptions

XCP_OUT_OF_MEMORY	if *buf_ascii.size* < *buf_ascii.len* + *len_append*

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>
void app(void)
{
    BUF_Object buf_ascii ;
    buf_ascii.data = "1111111111111111111111111111111";
    buf_ascii.size = 25 ;
    buf_ascii.len = 3 ;
    DEV_open(DEV_DEBUG);
    STR_append_itoa (678,&buf_ascii,3);
    DBG_CR;
    DBG_STR("buf_ascii (= ? 31 31 31 36 37 38)= ");
    DBG_DUMP(&buf_ascii);
    DEV_close(DEV_DEBUG);
}
```

3.14.5 STR_atoi

Converts the string contained in *buf_ascii* to an unsigned integer. The returned value is the converted value. This function reads the characters beginning at *buf_ascii->data* address and makes a left-right conversion. This continues until a non convertible character is encountered (+ or - included) or *buf_ascii->len* characters are read. If a non convertible character has been encountered, this function returns 0.

unsigned int STR_atoi (BUF_Handle *buf_ascii)***;**

Return Value

converted value

Parameters

input	*buf_ascii*	input buffer

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>
void app(void)
{
   unsigned int nbr ;
   BUF_Object buf_ascii = {3,10,0};
   buf_ascii.data = "100";
   DEV_open(DEV_DEBUG);
   DBG_CR;
   DBG_STR("buf_ascii = ");
   DBG_DUMP(&buf_ascii);
   nbr = STR_atoi(&buf_ascii);
   DBG_CR;
   DBG_STR("number = ");
   DBG_INT(nbr);
   DEV_close(DEV_DEBUG);
}
```

3.14.6 STR_blank

Fill the buffer buf with *buf->len* characters space (' ').

void STR_blank (BUF_Handle *buf***);**

Return Value

none

Parameters

input	*buf*	buffer to be filled with spaces
output	*buf*	buffer filled with buf->len spaces

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>
void app(void)
{
   BUF_Object buf = {20,20,0};
   char str[20];
   buf.data=str;
   DEV_open(DEV_DEBUG);
   STR_blank(&buf);
   DBG_CR;
```

```
    DBG_STR("buf blank = ");
    DBG_DUMP(&buf);
    DEV_close(DEV_DEBUG);
}
```

3.14.7 STR_compare

Compare the string specified by *buf1* to the string specified by *buf2*. The strings are compared, beginning at the given addresses, character by character, up to the length of the shorter string or until a difference is found. If the two strings are identical, the returned value is zero. If the two strings are identical up to the length of the shorter string, the returned value is -1. If the length of *buf1* is less than the length of *buf2* and 1 otherwise. If the two strings are not identical up to the length of the shorter string, num is -1 if the first non-matching character in the string specified by *buf1* has a lesser numeric value than the corresponding character in the string specified by *buf2* and 1 otherwise.

int STR_compare (BUF_Handle *buf1***, BUF_Handle** *buf2***);**

Return Value

0 if both buffers are identical

-1 if (*buf1->len)* < (*buf2->len)* or if firstval(*buf1)* < firstval(*buf2)*

1 if (*buf1->len)* > (*buf2->len)* or if firstval(*buf1)* > firstval(*buf2),*

firstval being the value of the first non-matching character

Parameters

input	*buf1*	first member of comparison
	buf2	second member of comparison

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>
void app(void)
{
   int i;
   BUF_Object buf1 = {0,20,0};
   BUF_Object buf2= {0,20,0};
   char str1[20],str2[20];
   buf1.data = str1;
   buf2.data = str2;
   DEV_open(DEV_DEBUG);
   BUF_set_str(&buf1,"STRING2");
   BUF_set_str(&buf2,"STRING2");
   i = STR_compare(&buf1,&buf2);
   DBG_CR;
   DBG_STR("Compare (? 0 )= ");
   DBG_INT(i);
   DEV_close(DEV_DEBUG);
}
```

3.14.8 STR_erase

Clear all bits in each of the *buf->len* consecutive address units of the buffer. Equivalent to STR_fill (buf, '\0');

void STR_erase (BUF_Handle *buf***);**

Return Value

none

Parameters

input	*buf*	buffer to be erased

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>
void app(void)
{
   BUF_Object buf = {20,20,0};
   char str[20];
   buf.data=str;
   DEV_open(DEV_DEBUG);
   STR_fill(&buf,'X');
   DBG_CR;
   DBG_STR("buf before erase = ");
   DBG_DUMP(&buf);
   STR_erase(&buf);
   DBG_CR;
   DBG_STR("buf after erase = ");
   DBG_DUMP(&buf);
   DEV_close(DEV_DEBUG);
}
```

3.14.9 STR_fetch_bcd

Convert to an unsigned integer the binary coded decimal string represented in the data field of *buf*.

unsigned int STR_fetch_bcd (BUF_Handle *buf***);**

Return Value

result of the conversion

Parameters

input	*buf*	binary coded decimal number string

Exceptions

XCP_DIGIT_TOO_LARGE	if a nibble is not a valid BCD digit

Example

```
#include <buf.h>
```

```
#include <str.h>
#include <dbg.h>

void app(void)
{
   unsigned int bcd ;
   BUF_Object buf_bcd = {4,4,0};
   char str_bcd[4];
   buf_bcd.data = str_bcd ;
   DEV_open(DEV_DEBUG);
   STR_store_bcd(10000,&buf_bcd);
   bcd = STR_fetch_bcd(&buf_bcd);
   DBG_CR;
   DBG_STR("FETCH bcd = ");
   DBG_INT(bcd);
   DEV_close(DEV_DEBUG);
}
```

3.14.10 STR_fetch_bn

Convert to an unsigned integer the binary number string represented in the data field of *buf*.

unsigned int STR_fetch_bn (BUF_Handle *buf***);**

Return Value

result of the conversion

Parameters

input	*buf*	binary number string

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>

void app(void)
{
   unsigned int bn ;
   BUF_Object buf_bn = {4,4,0} ;
   char str_bn[4];
   buf_bn.data = str_bn ;
   DEV_open(DEV_DEBUG);
   STR_store_bn(10000,&buf_bn);
   bn = STR_fetch_bn(&buf_bn);
   DBG_CR;
   DBG_STR("bn = ");
   DBG_INT(bn);
   DEV_close(DEV_DEBUG);
}
```

3.14.11 STR_fetch_cn

Convert the CN (compressed numeric) string represented in the data field of *buf_cn* to an ASCII numeric string represented in the data field of *buf_ASCII*.

void STR_fetch_cn (BUF_Handle *buf_cn* **, BUF_Handle** *buf_ASCII***);**

Return Value

none

Parameters

input	*buf_cn*	Compressed numeric string
output	*buf_ASCII*	ASCII numeric string

Exceptions

XCP_DIGIT_TOO_LARGE	a character in the CN string is not a number in the current base

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>
void app(void)
{
   char str_ASCII[10];
   char str_cn[10];
   int i;
   BUF_Object buf_ASCII = {0,10,0};
   BUF_Object buf_cn = {4,10,0};
   buf_ASCII.data = str_ASCII ;
   buf_cn.data = str_cn ;
   DEV_open(DEV_DEBUG);
   BUF_set_str(&buf_ASCII, "10000");
   STR_store_cn(&buf_ASCII,&buf_cn);   // buf_cn.len MUST be known before
   for(i=0; i<=9; str_ASCII[i++] = 0);
   STR_fetch_cn(&buf_cn,&buf_ASCII);   // buf_ascii contains "10000" again
   DBG_CR;
   DBG_STR("buf_ASCII (? 31 30 30 30 = 1000 )= ");
   DBG_DUMP(&buf_ASCII);
   DEV_close(DEV_DEBUG);
}
```

3.14.12 STR_fill

Store *c* in each of the *buf->len* consecutive characters of the buffer.

void STR_fill (BUF_Handle *buf,* **char** *c***);**

Return Value

none

Parameters

input	*buf*	buffer to be filled
	c	filler character
output	*buf*	buffer filled with the filler character

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>
void app(void)
{
   BUF_Object buf = {20,20,0};
   char str[20];
   buf.data=str;
   DEV_open(DEV_DEBUG);
   STR_fill(&buf,'X');
   DBG_CR;
   DBG_STR("buf (? 20 * 58)= ");
   DBG_DUMP(&buf);
   DEV_close(DEV_DEBUG);
}
```

3.14.13 STR_itoa

Convert the unsigned integer *num* to an ASCII string and store the string in *buf_ascii*.

void STR_itoa (BUF_Handle *buf_ascii*, **unsigned long** *num***);**

Return Value

none

Parameters

input	*num*	input number
output	*buf_ascii*	output buffer to store ASCII string

Exceptions

XCP_OUT_OF_MEMORY	if *buf_ascii* does not hold enough place to store the ASCII string

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>
void app(void)
{
   BUF_Object buf_ascii = { 0, 25, 0 };
   char str_ascii[25];
   DEV_open(DEV_DEBUG);
   STR_itoa (&buf_ascii,60784);
   DBG_CR;
   DBG_STR("buf_ascii (= ? 36 30 37 38 34)= ");
   DBG_DUMP(&buf_ascii);
```

```
    DEV_close(DEV_DEBUG);
}
```

3.14.14 STR_length

Return the number of characters of the null-terminated string *str.*

int STR_length (char ***str*);**

Return Value

length of the string

Parameters

input	*str*	string whose length is to be counted

Example

```
#include <str.h>
#include <mem.h>
#include <dbg.h>
void app(void)
{
   int i;
   char str[10];
   DEV_open(DEV_DEBUG);
   MEM_move(str,"hello",5);
   str[5] = 0;
   i = STR_length(str);
   DBG_CR;
   DBG_STR("length (? 5 )= ");
   DBG_INT(i);
   DEV_close(DEV_DEBUG);
}
```

3.14.15 STR_minus_trailing

Remove the trailing spaces from *buf* by reducing its length by the number of spaces at the end.

void STR_minus_trailing (BUF_Handle *buf***);**

Return Value

none

Parameters

input	*buf*	string to be shortened

Example

```
#include <buf.h>
#include <str.h>
#include <mem.h>
#include <dbg.h>
void app(void)
{
```

```
    int i;
    BUF_Object buf = {20,20,0};
    char str[20];
    DEV_open(DEV_DEBUG);
    for(i=0; i<=19; str[i++] = ' ');
    MEM_move(str, "STRING1", 7);
    buf.data=str;
    DBG_CR;
    DBG_STR("buf before minus_trailing = ");
    DBG_DUMP(&buf);
    STR_minus_trailing(&buf);
    DBG_CR;
    DBG_STR("buf (? STRING1)= ");
    DBG_DUMP(&buf);
}
```

3.14.16 STR_minus_zeros

Remove the trailing zeroes from *buf* by reducing its length by the number of zeroes at the end.

void STR_minus_zeros (BUF_Handle *buf***);**

Return Value

none

Parameters

input	*buf*	string to be shortened

Example

```
#include <buf.h>
#include <str.h>
#include <mem.h>
#include <dbg.h>
void app(void)
{
    int i;
    BUF_Object buf = {20,20,0};
    char str[20];
    DEV_open(DEV_DEBUG);
    for(i=0; i<=19; str[i++]=0);
    MEM_move(str, "STRING1", 7);
    buf.data=str;
    DBG_CR;
    DBG_STR("buf before minus_zeros = ");
    DBG_DUMP(&buf);
    STR_minus_zeros(&buf);
    DBG_CR;
    DBG_STR("buf (? STRING1)= ");
    DBG_DUMP(&buf);
    DEV_close(DEV_DEBUG);
}
```

3.14.17 STR_scan

Parse the character string (*buf*) for bytes containing *character.* The returned value is the

address where the first matching character was found or the end of string, *len* is the length of the remaining string or 0 if *character* was not found.

char *STR_scan (BUF_Handle *buf***, char** *character***, int ***len***);**

Return Value

address where the first matching character was found

Parameters

input	*buf*	input buffer
	character	character to search
output	*len*	len of the remaining string

Example

```
#include <str.h>
#include <mem.h>
#include <dbg.h>
void app(void)
{
   BUF_Object buf = {20,20,0};
   char str[20], *str_queue;
   int len, i;
   buf.data=str;
   DEV_open(DEV_DEBUG);
   for(i=0;i<=19;str[i++]=0);
   MEM_move(str+15, "MEM", 3);
   str_queue= STR_scan(&buf,'M',&len);
   DBG_CR;
   DBG_STR("queue = ");
   DBG_STR(str_queue);
   DBG_CR;
   DBG_STR("len = ");
   DBG_INT(len);
   DEV_close(DEV_DEBUG);
}
```

3.14.18 STR_skip

Parse the character string *str1*, skipping bytes that contain character *c*. The function returns a pointer to the address of the first byte that differs from character *c*. The function works only with null-terminated strings.

char *STR_skip(char **str1***, **char** *c***);**

Return Value

address of the first byte that differs from character *c*

Parameters

input	*string1*	string
	c	character to skip
output	*char**	rest of the string after skipping the given character

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>
void app(void)
{
   char str[20], *str_queue;
   int len, i;
   DEV_open(DEV_DEBUG);
   strcpy(str, "1111TEST");
   str_queue= STR_skip(str,'1');
   DBG_CR;
   DBG_STR("Test of skipping 1 from '");
   DBG_STR(str);
   DBG_STR("' gives (TEST) = '");
   DBG_STR(str_queue);
   DBG_STR_CR("'");
   DEV_close(DEV_DEBUG);
}
```

3.14.19 STR_store_bcd

Convert the number *val* to a string of type N (binary nibbles, padded with leading zeroes) and store it in the data field of *buf*, padding the string with leading zeroes on the length *buf->len*. If *buf->len* is shorter than the length of the string to be written, the result is left truncated.

void STR_store_bcd (unsigned int *val***, BUF_Handle** *buf***);**

Return Value

none

Parameters

input	*val*	value to be converted
output	*buf*	binary coded decimal number string

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>

void app(void)
{
   BUF_Object buf_bcd = {4,4,0};
   char str_bcd[4];
   buf_bcd.data = str_bcd ;
   DEV_open(DEV_DEBUG);
   STR_store_bcd(10000,&buf_bcd);  // result in buf_bcd.data : 00 01 00 00
```

```
    DBG_CR;
    DBG_STR("buf (? 00 01 00 00 = )= ");
    DBG_DUMP(&buf_bcd);
    DEV_close(DEV_DEBUG);
}
```

3.14.20 STR_store_bn

Convert the number *val* to a string of type BN (binary number) and store it in the data field of *buf*, padding the string with leading zeroes on the length *buf->len*. If *buf->len* is shorter than the length of the string to be written, the result is left truncated.

void STR_store_bn (unsigned int *val***, BUF_Handle** *buf***);**

Return Value

none

Parameters

input	*val*	value to be converted
output	*buf*	binary number string

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>

void app(void)
{
   BUF_Object buf_bn = {4,4,0} ;
   char str_bn[4];
   buf_bn.data = str_bn ;
   DEV_open(DEV_DEBUG);
   STR_store_bn(10000,&buf_bn); // result in buf_bn.data : 00 00 27 10
   DBG_CR;
   DBG_STR("buf (? 00 00 27 10 = )= ");
   DBG_DUMP(&buf_bn);
   DEV_close(DEV_DEBUG);
}
```

3.14.21 STR_store_cn

Convert the ASCII number string contained in *buf_ascii* to a string of type CN (Compressed Numeric) and store it in the data field of *buf_cn*, padding the string with trailing F's on the length *buf_cn->len*. If *buf_cn->len* is shorter than the length of the string to be written, the result is right truncated.

void STR_store_cn (BUF_Handle *buf_ascii***, BUF_Handle** *buf_cn***);**

Return Value

none

Parameters

input	*buf_ASCII*	ASCII numeric string to be converted
output	*buf_cn*	compressed numeric string

Exceptions

XCP_DIGIT_TOO_LARGE	a character in the ASCII string is not a number in the current base

Example

```
#include <buf.h>
#include <str.h>
#include <dbg.h>
void app(void)
{
   char str_ASCII[10];
   char str_cn[10];
   BUF_Object buf_ASCII = {0,10,0};
   BUF_Object buf_cn = {4,10,0};
   buf_ASCII.data = str_ASCII ;
   buf_cn.data = str_cn ;
   DEV_open(DEV_DEBUG);
   BUF_set_str(&buf_ASCII, "10000");
   STR_store_cn(&buf_ASCII,&buf_cn);  // result : 10 00 0F FF
   DBG_CR;
   DBG_STR("buf_cn (? 10 00 0F FF = )= ");
   DBG_DUMP(&buf_cn);
   DEV_close(DEV_DEBUG);
}
```

3.15 TLV services

	C function	Main Token invoked
3.15.1	TLV_bit_clear	TLVBITCLEAR
3.15.2	TLV_bit_fetch	TLVBITFETCH
3.15.3	TLV_bit_set	TLVBITSET
3.15.4	TLV_clear	TLVCLEAR
3.15.5	TLV_fetch	TLVFETCH
3.15.6	TLV_fetchraw	TLVFETCHRAW
3.15.7	TLV_find	TLVFIND
3.15.8	TLV_format	TLVFORMAT
3.15.9	TLV_get	TLVFETCH
3.15.10	TLV_get_length	TLVFETCHLENGTH
3.15.11	TLV_get_tag	TLVFETCHTAG
3.15.12	TLV_initialise	TLVINIT

Table 18: TLV functions

	C function	Main Token invoked
3.15.13	TLV_parse	TLVPARSE
3.15.14	TLV_parsed	TLVSTATUS
3.15.15	TLV_plusdol	TLVPLUSDOL
3.15.16	TLV_plusstring	TLVPLUSSTRING
3.15.17	TLV_put	TLVSTORE
3.15.18	TLV_status	TLVSTATUS
3.15.19	TLV_store	TLVSTORE
3.15.20	TLV_storeraw	TLVSTORERAW
3.15.22	TLV_tag	TLVTAG
3.15.22	TLV_traverse	TLVTRAVERSE

Table 18: TLV functions

TLV definitions

Note that there is no function to define TLVs. TLVs are defined in a separate source file with a ".tlv" extension (see Section 2.3 of this manual and the OTC C token compiler manual).

3.15.1 TLV_bit_clear

Clear (to zero) bit *bit_nr* in the data assigned to the TLV definition whose access parameter is *tag*. *bit_nr* = (bit position) + 8 * (byte position); *bit_nr* = 0 for LSB (least significant bit) of Byte 1.

An ambiguous condition exists if the TLV is not of type `TLV_B` or if *bit_nr* is larger than the bits contained in the assigned data.

void TLV_bit_clear (int *tag***, unsigned int** *bit_nr***);**

Return Value

none

Parameters

input	*tag*	TLV tag number
	bit_nr	bit position in the word, 0 being the least significant bit of the last (rightmost) byte

Exceptions

`XCP_UNDEFINED_TLV`	the tag is not associated with any known TLV

Example:

```
#include <tlv.h>
#define TLV_PRIM3 (int)(0x9f47)
TLV_bit_clear(TLV_PRIM3,5);  // clear the fifth bit of the value field
```

3.15.2 TLV_bit_fetch

Return the status of bit *bit_nr* in the data assigned to the TLV definition whose access parameter is *tag*. *bit_nr* = (bit position) + 8 * (byte position); *bit_nr* = 0 for LSB (least significant

bit) of Byte 1. An ambiguous condition exists if the TLV is not of type TLV_B or if *bit_nr* is larger than the bits contained in the assigned data.

unsigned int TLV_bit_fetch (int *tag*, unsigned int *bit_nr*);

Return Value

Return -1 if the referenced bit was set. Otherwise, 0 is returned.

Parameters

input	*tag*	TLV tag number
	bit_nr	bit position in the word, 0 being the least significant bit of the last (rightmost) byte

Exceptions

XCP_UNDEFINED_TLV	the tag is not associated with any known TLV

Example:

```
#include <tlv.h>
#include <dbg.h>
#define TLV_PRIM3 (int)(0x9f47)

if (TLV_bit_fetch(TLV_PRIM3,5) != 0) {
   i = 1;
}
else {
   i = 0;
}
DBG_STR("Bit 5 of PRIM3: "); DBG_INT(i);
```

3.15.3 TLV_bit_set

Set (to one) bit *bit_nr* in the data assigned to the TLV definition whose access parameter is *tag*. *bit_nr* = (bit position) + 8 * (byte position); *bit_nr* = 0 for LSB (least significant bit) of Byte 1.

An ambiguous condition exists if the TLV is not of type TLV_B or if *bit_nr* is larger than the bits contained in the assigned data.

void TLV_bit_set (int *tag*, unsigned int *bit_nr*);

Return Value

none

Parameters

input	*tag*	TLV tag number
	bit_nr	bit position in the word, 0 being the least significant bit of the last (rightmost) byte

Exceptions

XCP_UNDEFINED_TLV	the tag is not associated with any known TLV

Example:

```
#include <dbg.h>
#include <tlv.h>

#define TLV_PRIM3  (int)(0x9f47)

TLV_bit_set(TLV_PRIM3,5);  // Set to one the fifth bit of the value field
```

3.15.4 TLV_clear

For the TLV whose access parameter is *tag*, clear the assigned status bit and remove any assignment of data associated with the TLV definition.

void TLV_clear(int *tag***);**

Return Value

none

Parameters

input	*tag*	TLV tag number

Exceptions

XCP_UNDEFINED_TLV	the tag is not associated with any known TLV

Example:

```
#include <tlv.h>
#include <dbg.h>
#define TLV_BINARY_NBR (int)(0x9F5B)

int i;
i = TLV_status (TLV_BINARY_NBR);    // see 3.15.18
DBG_STR("Status before TLV_clear : ");
DBG_INT(i);
TLV_clear(TLV_BINARY_NBR);
i = TLV_status (TLV_BINARY_NBR);
DBG_STR("Status after TLV_clear : ");
DBG_INT(i);
```

3.15.5 TLV_fetch

Fetch a binary number (TLV_BN) or BCD value (TLV_N) assigned to the TLV definition whose tag number is *tag*. Apply a conversion according to the associated type field. Return an unsigned number.

unsigned int TLV_fetch (int *tag***);**

Return Value

value of the TLV, if the tlv is not parsed 0 is returned

Parameters

input	*tag*	TLV tag number

Exceptions

XCP_UNDEFINED_TLV	the tag is not associated with any known TLV
XCP_ARG_TYPE_MISMATCH	the TLV holds neither a binary number (TLV_BN) nor BCD value (TLV_N)

Example:

```
#include <dbg.h>
#include <tlv.h>
#define TLV_BINARY_NBR (int)(0x9F5B)

int tlv_numeric_value;
tlv_numeric_value = TLV_fetch(TLV_BINARY_NBR);
DBG_STR("TLV fetch BINARY NBR : ");
DBG_INT(tlv_numeric_value); DBG_CR;
```

3.15.6 TLV_fetchraw

Put the data assigned to the TLV definition whose access parameter is *tag* in the buffer pointed by *buf* without applying any conversion. The format of the data fetched is the same as the value field format of the corresponding TLV. The length returned (*buf->len*) for a TLV which has no data assigned to it shall be 0.

int TLV_fetchraw (int *tag***, BUF_Handle** *buf***);**

Return Value

0 if TLV not parsed, otherwise -1

Parameters

input	*tag*	TLV tag number
output	*buf*	buffer containing the TLV without any type conversion

Exceptions

XCP_UNDEFINED_TLV	the tag is not associated with any known TLV

Example

```
#include <tlv.h>
#include <dbg.h>
#define TLV_ASCII (int)(0x9F60)

char str[256];
BUF_Object buf = { 0, 256; 0 };
buf.data = str;
TLV_fetchraw(TLV_ASCII, &buf);
DBG_STR ("TLV_ASCII RAW : ");
DBG_DUMP(&buf_data); DBG_CR;
```

3.15.7 TLV_find

Return the address of the TLV definition whose tag number is *tag*. If there is no TLV defined for tag *tag*, return zero.

tlv_t * TLV_find (int *tag***);**

Return Value

TLV address

Parameters

input	*tag*	TLV tag number

Example:

```
#include <tlv.h>
#include <dbg.h>
#define TLV_PRIM1 (int)(0x9f46)

// illustrates how to access tlv's address
tlv_t *tlv_ptr ;
int tlv_tag ;
tlv_ptr = TLV_find(TLV_PRIM1);
DBG_STR("TLV PTR : "); DBG_INT((int)tlv_ptr);
// illustrates how to access tlv's tag given it's address
tlv_tag = TLV_tag(tlv_ptr) ;
DBG_STR("TLV TAG T: "); DBG_INT(tlv_tag);
```

3.15.8 TLV_format

Return the format code associated with the TLV definition whose access parameter is *tag*.

enum TLV_type TLV_format (int *tag***);**

Return Value

- `TLV_N` for n data, BCD nibbles padded with leading zeroes.
- `TLV_B` for b data, sequences of application-defined bytes.
- `TLV_BN` for bn data, a binary number.
- `TLV_CN` for cn data, a compressed numeric which is BCD nibbles padded with trailing F's.
- `TLV_AN` for an data, alphabetic and numeric characters, with trailing zeroes.
- `TLV_ANS` for ans data, the full ASCII character set, with trailing zeroes.
- `TLV_VAR` for var data, data of variable format.

Parameters

input	*tag*	TLV tag number

Exceptions

`XCP_UNDEFINED_TLV`	the tag is not associated with any known TLV

Example

```
#include <tlv.h>
#include <dbg.h>
#define TLV_ASCII (int)(0x9F60)

int tlv_type;
tlv_type = TLV_format(TLV_ASCII);
```

3.15.9 TLV_get

Fetch the data associated with the TLV whose access parameter is *tag*, write it into the global character buffer `TLV_value` and set the global integer variable `TLV_length` to the length of the data fetched.

int TLV_get (int *tag***);**

Return Value

-1 if tlv parsed, otherwise 0

Parameters

input	*tag*	TLV tag number

Remarks

- This function cannot be used to fetch values of the `TLV_BN` or `TLV_N` format; for that, `TLV_fetch` must be used.
- The byte following the last byte of data written to `TLV_value` is set to zero by `TLV_get` so that, if needed, the value can be treated as a null-terminated string.

Exceptions

`XCP_UNDEFINED_TLV`	the tag is not associated with any known TLV
`XCP_ARG_TYPE_MISMATCH`	the TLV holds a binary number (type `TLV_BN`) or BCD value (type `TLV_N`)

Example:

```
#include <tlv.h>
#include <dbg.h>
#define TLV_ASCII (int)(0x9F60)

TLV_get(TLV_ASCII);
DBG_STR(" length = ");
DBG_INT(TLV_length);      // TLV_length is a global variable
DBG_STR(" value = ");
DBG_STR(TLV_value);       // TLV_value is a global variable
// NB: TLV_value can be printed because it is a null terminated string
```

3.15.10 TLV_get_length

Parse the input buffer for a length field. The char** *ptr* contains the address of the first character of the ‘value’ field of the TLV. *lenV* is the length given in the input buffer minus the size of the length field.

int TLV_get_length (BUF_Handle *buf_lv***, char**** *ptr***, int ***lenV***);**

Return Value

value of the length field

Parameters

input	*buf_lv*	buffer to be parsed
output	*ptr*	pointer to the address of the string after the length

	lenV	pointer to the length of the string after the length field

Example:

```
#include <dbg.h>
#include <buf.h>
#include <tlv.h>
...
BUF_Object buf_LV = {0,117,0} ;
...
buf_LV.len = tlv_len ;
buf_LV.data = tlv_value_ptr ;
DBG_STR("Constructed buffer without TAG : ");
DBG_DUMP(&buf_LV); DBG_CR;
// TLV_get_length should always be called after TLV_get_tag
TLV_get_length(&buf_LV, &tlv_value_ptr1, &tlv_len);
DBG_STR ("Length of 'Value' string : "); DBG_INT (tlv_len);
DBG_STR ("Address of 'Value' string : "); DBG_INT ((U16)tlv_value_ptr1);
```

3.15.11 TLV_get_tag

Parse the input buffer for a tag field. The function returns the decoded tag value. The char** points to the byte immediately following the tag, and *lenLV* points to an integer containing the original length minus the length of the tag.

int TLV_get_tag (BUF_Handle *buf_tlv***, char**** *ptr***, int ***lenLV***);**

Return Value

value of the tag found

Parameters

input	*buf_tlv*	buffer to be parsed
	ptr	pointer to the address of the string after the tag
	lenLV	pointer to the length of the string after the tag (including length of the length field)

Example:

```
#include <tlv.h>
#include <dbg.h>

char *tlv_value_ptr ;
int tlv_len, tlv_tag ;
// How to get length given a tlv buffer ?
tlv_tag = TLV_get_tag(&buf_constructed, &tlv_value_ptr,&tlv_len);
   // (see example in 3.15.13 for definition)
DBG_STR ("Tag of the constructed tlv : "); DBG_INT (tlv_tag);
DBG_STR ("Length of the 'LV' string : "); DBG_INT (tlv_len);
DBG_STR ("Address of the LV string : "); DBG_INT ((U16)tlv_value_ptr);
```

3.15.12 TLV_initialise

Clear all internally-maintained data associated with TLV definitions and set the status of all TLV definitions to not assigned. This command is provided to satisfy the security requirement that a financial application must clear all transaction related data after the transaction

has been completed.

void TLV_initialise (void);

Return Value

none

Parameters

none

Example:

```
#include <tlv.h>

TLV_initialise();
```

3.15.13 TLV_parse

Process the given buffer for TLV sequences, expecting a valid combination of primitive or constructed TLV definitions. For each TLV encountered in the string, the value field is assigned to the TLV definition which is associated with that tag. For each successfully parsed TLV, the assigned status bit in the definition record shall be set. If a tag is not found no exception is thrown. When a constructed TLV is encountered, whether or not its tag is found, its value field shall be recursively parsed for embedded TLV sequences.

void TLV_parse (BUF_Handle *buf***);**

Return Value

none

Parameters

input	*buf*	string of TLVs to parse

Remarks

- The TLV data structure must be filled at module initialisation, type and value must be consistent,
- if the tag is not associated with any known TLV, no error is generated and parsing continues.

Exceptions

XCP_STRING_TOO_LARGE	the TLV strng is not correctly formatted

Example:

```
#include <tlv.h>
#include <dbg.h>
#include <buf.h>

#define TLV_PRIM1  (int)(0x9f46)  // part of TLV_CONSTRUCTED
#define TLV_PRIM2  (int)(0x9f48)  // part of TLV_CONSTRUCTED
#define TLV_PRIM3  (int)(0x9f47)  // part of TLV_CONSTRUCTED
#define TLV_CONSTRUCTED    (int)(0x70)    // recursive tlv
```

```
BUF_Object buf_constructed = { 118,118, 0 };
char str_constr[118]={
   0x70, 0x74,
      // ICC public key certificate (9f63v2,9f46v3)
      0x9F,0x46, 0x44,
         0x79, 0xF3, 0x2B, 0xC8, 0x7F, 0x34, 0x98, 0xBF, 0x9A, 0x40,
         0x10, 0x71, 0xEB, 0xDB, 0x87, 0x94, 0x70, 0x8F, 0xA4, 0xDE,
         0x7D, 0x04, 0x45, 0x2F, 0x7A, 0x0B, 0xAA, 0x8C, 0x46, 0x1B,
         0x74, 0x66, 0x00, 0xED, 0x8A, 0x3C, 0xB9, 0x5D, 0xF0, 0x53,
         0xD7, 0x9D, 0x29, 0x37, 0xB8, 0xBF, 0x58, 0x91, 0xCC, 0x0A,
         0xBF, 0x98, 0x09, 0x8E, 0x37, 0x0A, 0x16, 0x9D, 0x86, 0xE1,
         0xE9, 0x73, 0xBD, 0x05, 0x0F, 0x0B, 0x45, 0x28,
      // ICC public key remainder (9f64v2,9f48v3)
      0x9F,0x48,  0x26,
         0x7B, 0x52, 0xEF, 0x93, 0x0D, 0x10, 0x59, 0x57, 0x11, 0x1E,
         0x3D, 0x70, 0x01, 0x93, 0xFA, 0x8E, 0x26, 0xAC, 0xD6, 0x68,
         0xF1, 0x08, 0xE1, 0xCF, 0x25, 0x2B, 0x6B, 0xBB, 0x9B, 0x65,
         0xA9, 0x6F, 0xF7, 0xAE, 0xE6, 0x8E, 0xEF, 0xAF,
      // ICC public key exponent (9f65v2,9f47v3)
      0x9F,0x47,  0x01,
         0x03};
buf_constructed.data = str_constr ;

TLV_parse(&buf_constructed);
```

3.15.14 TLV_parsed

Return the parsed bit (last bit) of the status flag associated with the given TLV.

unsigned int TLV_parsed (int *tag***);**

Return Value

not parsed = 0
parsed = 1

Parameters

input	*tag*	TLV tag number

Exceptions

XCP_UNDEFINED_TLV	the tag is not associated with any known TLV

Example

```
#include <tlv.h>
#include <dbg.h>
#define TLV_ASCII          (int)(0x9F60)

int tlv_status;
tlv_status = TLV_parsed(TLV_ASCII);
DBG_STR ("TLV_parsed : ");
DBG_INT(tlv_status);
MEM_move(TLV_value,"Hello",5);
TLV_length = 5;
TLV_put(TLV_ASCII);
tlv_status = TLV_parsed(TLV_ASCII);
```

```
DBG_STR ("TLV_parsed : ") ;
DBG_INT(tlv_status);
```

3.15.15 TLV_plusdol

Process the input buffer *buf_dol* for a sequence of tag and length fields. For all tag fields encountered, the corresponding value fields are inserted into the buffer *buf_data* up to its maximum size. The parts of the resulting string are filled or truncated where a specified length field differs from the size of the existing value; see *EMV Integrated Circuit Card Specification for Payment Systems*, Part 2 (Data Elements and Commands).

void TLV_plusdol (BUF_Handle *buf_dol***, BUF_Handle** *buf_data***);**

Return Value

none

Parameters

input	*buf_dol*	Data Object List (TL+TL+...+TL) used to build the constructed list
output	*buf_data*	resulting constructed list (V+V+...+V)

Exceptions

XCP_STRING_TOO_LARGE	the buf_data is too small for the list to be constructed

Example:

```
#include <dbg.h>
#include <tlv.h>
#include <buf.h>

// illustrates how to fill in a Data List (with tlv's value fields)
// given a Data Object List (list of tag-length values)
// overwrites previous buf_data value, updates the length field of
// buf_data
// See example of TLV_parse for the contents of the string to be
// constructed.
BUF_Object buf_dol = {6,10,0};
BUF_Object buf_data = {0,150,0} ;
char str_dol[6]={ 0x9F,0x46, 0x44,0x9F,0x48,0x26 } ;
char str_data[150];
buf_dol.data = str_dol ;
buf_data.data = str_data ;
DBG_STR ("Data List length (before TLV plusdol) = ");
i = buf_data.len;
buf_data.len= 118;
DBG_INT(i);
DBG_STR("Dol buffer: ");
DBG_DUMP(&buf_dol);
TLV_plusdol(&buf_dol,&buf_data);
DBG_STR ("Data List Length (after TLV plusdol) = ");
i = buf_data.len ;
DBG_INT (i);
DBG_STR("data: "); DBG_DUMP(&buf_data);
```

3.15.16 TLV_plusstring

Retrieve tag, length, and assigned value from the TLV definition whose tag is *tag*. Append a well-formed TLV sequence to the end of the output buffer *buf_data*. Append an empty TLV if no data has been assigned. It is the programmer's responsibility to ensure that there is sufficient room at the end of the output string to hold both strings.

void TLV_plusstring (int *tag***, BUF_Handle** *buf_data***);**

Return Value

none

Parameters

input	*tag*	TLV tag number
output	*buf_data*	resulting constructed string

Exceptions

`XCP_UNDEFINED_TLV`	the tag is not associated with any known TLV

Example:

```
BUF_Object buf_data = {0,150,0} ;
char str_data[150];
// see 3.15.13 for buf_constructed definition

buf_constructed.data = str_constr ;
TLV_plusstring(TLV_CONSTRUCTED,&buf_data);
DBG_STR(" TLV_plusstring(TLV_CONSTRUCTED, buf_data) before parsed :");
DBG_STR("buf_data=");  // buf_data contains tag & length only
DBG_DUMP(&buf_data);
...
TLV_parse(& buf_constructed);
TLV_plusstring(TLV_CONSTRUCTED,&buf_data);
DBG_STR(" TLV_plusstring(TLV_CONSTRUCTED, buf_data) after parsed :");
DBG_STR("buf_data=");  // buf_data contains tlv constructed
DBG_DUMP(&buf_data);
```

3.15.17 TLV_put

Store the data defined by the global character buffer `TLV_value` and the global integer `TLV_length` to the TLV record whose access parameter is *tag* and set the assigned status bit for this TLV.

void TLV_put (int *tag***);**

Return Value

none

Parameters

input	*tag*	TLV tag number

implicitly, the global variables: `TLV_value` and `TLV_length`

Remarks

This function must not be used to store values for `TLV_N` or `TLV_BN`; for that, `TLV_store` must be used.

Exceptions

`XCP_UNDEFINED_TLV`	the tag is not associated with any known TLV
`XCP_ARG_TYPE_MISMATCH`	the TLV holds a binary (`TLV_BN` type) or BCD value (`TLV_N` type)

Example:

```
#include <tlv.h>
#define TLV_ASCII (int)(0x9F60)

MEM_move(TLV_value,"Hello",5);
TLV_length = 5;
TLV_put(TLV_ASCII);
```

3.15.18 TLV_status

Return the status flag associated with the given TLV.

unsigned int TLV_status (int *tag***);**

Return Value

status flag (rightmost byte)

0	0 = value field not assigned, 1 = value field assigned
1-7	Reserved for future use

Parameters

input	*tag*	TLV tag number

Exceptions

`XCP_UNDEFINED_TLV`	the tag is not associated with any known TLV

Example:

```
#include <tlv.h>
#include <dbg.h>
#define TLV_BINARY_NBR     (int)(0x9F5B)

int i;
TLV_store(TLV_BINARY_NBR,(unsigned int)1418);
i= TLV_status (TLV_BINARY_NBR);
DBG_STR("Status before TLV_clear : "); DBG_INT(i);
TLV_clear(TLV_BINARY_NBR);
i= TLV_status (TLV_BINARY_NBR);
DBG_STR("Status after TLV_clear : "); DBG_INT(i);
```

3.15.19 TLV_store

Store the unsigned integer *value* into the TLV whose tag number is *tag* and set the ‘parsed’ bit

in the TLV status byte. This function must be used to store binary numbers (`TLV_BN`) or BCD values (`TLV_N`); for other types, `TLV_put` must be used.

void TLV_store (int *tag*, unsigned int *value*);

Return Value

none

Parameters

input	*tag*	TLV tag number
	value	value to be stored

Exceptions

`XCP_UNDEFINED_TLV`	the tag is not associated with any known TLV
`XCP_ARG_TYPE_MISMATCH`	the value is not of type `TLV_N` or `TLV_BN`

Example:

```
#include <tlv.h>
#define TLV_BINARY_NBR (int)(0x9F5B)

int tlv_numeric_value;
TLV_store(TLV_BINARY_NBR,(unsigned int)1418);
```

3.15.20 TLV_storeraw

Assign the data pointed by the *buf* structure, without applying any conversion, to the TLV definition whose access parameter is *tag*. The format of the data supplied must be the same as the value field format of the corresponding TLV. Same as TLV_put for a non-numeric TLV type.

void TLV_storeraw (int *tag*, BUF_Handle *buf*);

Return Value

none

Parameters

input	*tag*	TLV tag number
output	*buf*	buffer containing the TLV without any type conversion

Exceptions

`XCP_UNDEFINED_TLV`	the tag is not associated with any known TLV

Example

```
#include <tlv.h>
#include <dbg.h>
#define TLV_ASCII (int)(0x9F60)

BUF_Object buf_data = { 3, 3, 0 };
buf_data.data = "AAA";
TLV_storeraw(TLV_ASCII, &buf_data);
```

3.15.21 TLV_tag

Return the tag number for the TLV definition pointed by the *tlv* address.

int TLV_tag (tlv_t * *tlv***);**

Return Value

Tag number

Parameters

input	*tlv*	address of a TLV

Example:

See `TLV_find`.

3.15.22 TLV_traverse

Process *buf->len* bytes at *buf->data* for TLV sequences, expecting a valid combination of primitive or constructed TLV definitions. For each TLV encountered in the string, the tag and the whole of the value field is passed to the function *fn*. Exception -507 (`XCP_STRING_TOO_LARGE`) will be thrown if the tag, length and value fields are not entirely contained within the input string. When a constructed TLV is encountered, its tag and value field will be passed to the function *fn* and then the value field will be recursively traversed for embedded TLV sequences which are, in turn, passed to *fn*. The prototype of the function *fn* is required to be : `void fn(int tag, BUF_Handle buf_value);` where tag is the TLV's tag and *buf_value* is the value field.

void TLV_traverse (BUF_Handle *buf***, FUNC** *fn* **);**

Return Value

none

Parameters

input	*buf*	string of TLV's to traverse
	fn	function handling the individual TLV's

Exceptions

`XCP_STRING_TOO_LARGE`	the tag, length and value fields are not contained within the input string

3.16 Time Handling Services

	C function	Main Token invoked
3.16.1	TMR_clock	GETMS
3.16.2	TMR_date	GETTIME
3.16.3	TMR_delay	MS
3.16.4	TMR_expired	GETMS
3.16.5	TMR_set	GETMS
3.16.6	TMR_set_date	SETTIME
3.16.7	TMR_wait_expired	GETMS

Table 19: TMR functions

3.16.1 TMR_clock

Return the current value of the system milliseconds timer (number of milliseconds, between 1 and 2^31-1).

int TMR_clock (void);

Return Value

the current value of the system milliseconds timer

Parameters

none

Example:

```
#include <tmr.h>
#include <dbg.h>
void app(void)
{
   int clock;
   // Test TMR_clock : see the the current value of the system milliseconds
   // timer at two different moments
   DEV_open(DEV_DEBUG);
   clock = TMR_clock();
   DBG_INT(clock); DBG_CR;
   clock = TMR_clock();
   DBG_INT(clock); DBG_CR;
   DEV_close(DEV_DEBUG);
 }
```

3.16.2 TMR_date

Put the system time and date into a `TMR_date_t` type buffer.

void TMR_date (TMR_date_t ***date*);**

Return Value

none

Parameters

output	*date*	The current date and time.

Example:

```
#include <tmr.h>
#include <dbg.h>
void app(void)
{
   TMR_date_t date;
   DEV_open(DEV_DEBUG);
   TMR_date(&date);
   DBG_STR("Year :"); DBG_INT(date.year); DBG_CR;
   DBG_STR("Month :"); DBG_INT(date.month); DBG_CR;
   DBG_STR("Day :"); DBG_INT(date.day); DBG_CR;
   DBG_STR("Hour :"); DBG_INT(date.hour); DBG_CR;
   DBG_STR("Minute :"); DBG_INT(date.min); DBG_CR;
   DBG_STR("Second :"); DBG_INT(date.sec); DBG_CR;
   DEV_close(DEV_DEBUG);
}
```

3.16.3 TMR_delay

Wait for *ms* milliseconds.

void TMR_delay (int *ms***);**

Return Value

none

Parameters

input	*ms*	the time to wait in milliseconds

Example:

```
#include <tmr.h>
#include <dbg.h>
void app(void)
{
   // Test TMR_delay : wait for ms milliseconds
   DEV_open(DEV_DEBUG);
   DBG_STR("Hello - WAIT 5 SECONDS"); DBG_CR;
   TMR_delay(5000);
   DBG_STR("Bye"); DBG_CR;
   DEV_close(DEV_DEBUG);
}
```

3.16.4 TMR_expired

`TMR_expired` returns -1 if the timer *timer_nr* has expired.

int TMR_expired (int *timer_nr***);**

Return Value

-1 if the timer *timer_nr* is expired

Parameters

input	*timer_nr*	the reference number of this timer that you want to control if it's expired

Example:

See previous example (Section 3.16.5).

3.16.5 TMR_set

Set the start of timer *timer_nr* of *duration* milliseconds. The total number of timers you can use is limited to 8. The first one has a number 0.

void TMR_set (int *timer_nr***, int** *duration***);**

Return Value

none

Parameters

input	*timer_nr*	the reference number of this timer
	duration	the duration in milliseconds of the timer

Example:

```
#include <tmr.h>
#include <dbg.h>
void app(void)
{
   // Test TMR_set and TMR_expired : No is written until
   // the expiration is reached. When reached, Exp is written.
   int i;
   DEV_open(DEV_DEBUG);
   for (i=0;i<2;i++) {
      // Test TMR_set
      TMR_set(i, 3000*i);
      // Test TMR_expired : No is written until
      // the expiration was reached. When reached, Exp is writen.
      while (TMR_expired(i)!=-1) {
         DBG_STR(" No");
      }
      DBG_CR; DBG_STR(" Exp");
   }
   DEV_close(DEV_DEBUG);
}
```

3.16.6 TMR_set_date

Set a new date and time.

void TMR_set_date (TMR_date_t ****date*);**

Return Value

none

Parameters

input	*buf*	the current date and time

Example:

```
#include <tmr.h>
#include <dbg.h>
void app(void)
{
   // TMR_date and TMR_set_date : change the current date and time
   TMR_date_t date;
   DEV_open(DEV_DEBUG);
   TMR_date(&date);
   date.year = 1998;
   TMR_set_date(&date);
   TMR_date(&date);
   DBG_STR("NEW YEAR IS : ");
   DBG_INT(date.year); DBG_CR;
   DEV_close(DEV_DEBUG);
}
```

3.16.7 TMR_wait_expired

`TMR_wait_expired` waits until timer *timer_nr* is expired.

void TMR_expired (int *timer_nr***);**

Parameters

input	*timer_nr*	the reference number of this timer

Example:

```
#include <tmr.h>
#include <dbg.h>
void app(void)
{
   DEV_open(DEV_DEBUG);
   DBG_STR_CR("TMR_set to  4 seconds");
   TMR_set(1,4000);
   DBG_STR_CR("TMR_delay 2 sec");
   TMR_delay(2000);
   DBG_STR_CR("TMR_wait expired should wait 2 sec.:");
   TMR_wait_expired(1);
   DBG_STR_CR("Expired");
   DBG_STR_CR("TMR_set 5 sec.");
   TMR_set(2,5000);
   DBG_STR_CR("TMR_delay 6 sec");
   TMR_delay(6000);
   DBG_STR_CR("TMR_wait expired should not wait:");
   TMR_wait_expired(2);
   DBG_STR_CR("After TMR_wait_expired");
   DEV_close(DEV_DEBUG);
}
```

3.17 Variable access functions

Definition	C function	Main Token invoked
3.17.1	VAR_arg	PICK
3.17.2	VAR_start	DEPTH

Table 20: VAR functions

3.17.1 VAR_arg

Macro definition. Allows to access parameters in a function having a variable number of arguments. Retrieves the non explicitely declared argument *arg_nr* and assigns it the type *type*. Argument numbers begin at 0 (explicitely declared arguments not comprised). The arguments to retrieve must mandatory hold in one cell .

type VAR_arg(arg_nr, type);

Parameters

input	*arg_nr*	argument to retrieve
	type	type to give to this argument

Example

```
//   This entry point can have two prototypes:
//     int app('b', int arg1, BUF_Handle buf) when call_mode is 'b'
//     int app('i', int arg1, int arg2, int arg3 ) when call_mode is 'i'

int app(int call_mode,...)
{
   int arg1, arg2, arg3;
   BUF_Handle buf_arg2 ;

   DEV_open(DEV_DEBUG);
   VAR_start() ;

   switch ( call_mode ) {
      case 'i':
         arg1= VAR_arg(0,int);
         arg2= VAR_arg(1,int);
         arg3= VAR_arg(2,int);
         break;
      default :
         arg1= VAR_arg(0,int);
         buf_arg2= VAR_arg(1,BUF_Handle);
         break;
   }
   switch ( call_mode ) {
      case 'i':
         DBG_STR("First argument :"); DBG_INT(arg1); DBG_CR;
         DBG_STR("Second argument :"); DBG_INT(arg2); DBG_CR;
         DBG_STR("Third argument :"); DBG_INT(arg3); DBG_CR;
         return 24 ;
```

```
      default :
         DBG_STR("First argument :"); DBG_INT(arg1); DBG_CR;
         DBG_STR("Second argument :");
         DBG_DUMP(buf_arg2); DBG_CR;
         return 16 ;
   }
   DEV_close(DEV_DEBUG);
}
```

3.17.2 VAR_start

Initial statement needed when using `VAR_arg`. It must be used immediately after having declared the local variables.

void VAR_start(void);

Example

See VAR_arg

3.18 Exception functions

Definition	C function	Main Token invoked
3.18.1	XCP_arg_catch	CATCH
3.18.2	XCP_catch	CATCH
3.18.3	XCP_qthrow	QTHROW
3.18.4	XCP_throw	THROW

Table 21: XCP functions

3.18.1 XCP_arg_catch

Call the *fn* function in such a way that control can be transferred to a point just after this call if an `XCP_throw` is executed during the execution of the *fn* function. This function is the same as `XCP_catch` (see below) except that there is no restriction on the prototype of the function caught. If the function has a return value and completes normally, the value can be retrieved in the global variable `XCP_return_catch`.

int XCP_arg_catch (int *nbr_arg***, FUNC** *fn***,** *list of arguments...***);**

Return Value

return 0 if the *fn* function completes normally, or the throw code if an exception throw occurred during execution of *fn.*

Parameters

input	*nbr_arg*	number of arguments in fn
	fn	function to be catched
	list of arguments	arguments passed to fn

Example

```
#include <xcp.h>
#include <dbg.h>

int division3(unsigned char char1,unsigned char char2)
{
    XCP_qthrow(XCP_ZERO_DIVIDE,!char2);
    return char1/char2 ;
}

void app(void)
{
   DEV_open(DEV_DEBUG);
   DBG_STR(" Must catch 0: ");
   DBG_INT(XCP_arg_catch (2, (FUNC)division3, 4, 2)); DBG_CR;
   DBG_STR(" Result of the division(MUST be 2): ");
   DBG_INT(XCP_return_catch);DBG_CR;
   DBG_STR(" Must catch -10 (XCP_ZERO_DIVIDE): ");
   DBG_INT(XCP_arg_catch (2, (FUNC)division3, 2, 0)); DBG_CR;
   DEV_close(DEV_DEBUG);
}
```

3.18.2 XCP_catch

Call the *fn* function in such a way that control can be transferred to a point just after this call if an `XCP_throw` is executed during the execution of the *fn* function. If the function has a return value and completes normally, the value can be retrieved in the global variable `XCP_return_catch`.

int XCP_catch (FUNC *fn*);

Return Value

return 0 if the *fn* function completes normally, or the throw code if an exception throw occurred during execution of *fn*

Parameters

input	*fn*	called function

Remark

`XCP_catch` works only for functions without arguments.

Example

```
#include <xcp.h>
#include <dbg.h>
void level_2(void)
{
   DBG_STR("Hello"); DBG_CR;
   XCP_throw(-10);
}
void level_1(void)
{
   int throw_code;
   throw_code = XCP_catch(level_2);
```

```
   DBG_STR("Throw with sub-routine: code = ");
   DBG_INT(throw_code); DBG_CR;
}
void app(void)
{
   int throw_code;
   DEV_open(DEV_DEBUG);
   throw_code = XCP_catch(level_1);
   DBG_STR("No Throw found catch in level_1: code = ");
   DBG_INT(throw_code); DBG_CR;
   DEV_close(DEV_DEBUG);
}
```

3.18.3 XCP_qthrow

If *flag* is not equal to zero, this function throws with value *code*. If *flag* is zero, this function is a no-op.

void XCP_qthrow (int *code***, int** *flag***);**

Return Value

none

Parameters

input	*code*	exception code number
	flag	condition

Example

```
#include <xcp.h>
#include <dbg.h>
#define ZERO 0
#define NONZERO 1

void No_throw(void)
{
   XCP_qthrow(-20,ZERO);
}
void Throw(void)
{
   XCP_qthrow(-22,NONZERO);
}

void app(void){
   int throw_code;
   DEV_open(DEV_DEBUG);
   throw_code = XCP_catch(No_throw);
   DBG_STR("No Throw (QTHROW): code = "); // must return 0
   DBG_INT(throw_code); DBG_CR;
   throw_code = XCP_catch(Throw);
   DBG_STR("Must Throw (QTHROW): code = "); // must return -22
   DBG_INT(throw_code); DBG_CR;
   DEV_close(DEV_DEBUG);
}
```

3.18.4 XCP_throw

Raise an exception throw with value *code*. This value will be the returned value of the last executed catch.

void XCP_throw (int *code***);**

Return Value

none

Parameters

input	*code*	exception code number

Example

See `XCP_catch`

3.19 ANSI C Library Functions

	C function	Main Token invoked
3.19.1	memchr	SCAN
3.19.2	memcmp	COMPARE
3.19.3	memcpy	MOVE
3.19.4	memmove	MOVE
3.19.5	memset	FILL
3.19.6	sprintf	
3.19.7	strcat	PLUSSTRING
3.19.8	strchr	SCAN
3.19.9	strcmp	COMPARE
3.19.10	strcpy	MOVE
3.19.11	strlen	
3.19.12	strncat	PLUSSTRING
3.19.13	strncmp	COMPARE
3.19.14	strncpy	MOVE

Table 22: ANSI C Library Functions

3.19.1 memchr

The `memchr` function looks for the first occurrence of *c* in the first *len* bytes of *buf*. It stops when it finds *c* or when it has checked the first *len* bytes.

void *memchr(void **buf***, **char** *c*, **int** *len***);**

Return Value

If successful, memchr returns a pointer to the first location of *c* in *buf*. Otherwise, it returns `NULL`.

Parameters

input	*buf*	pointer to buffer
	c	character to look for
	len	number of characters

3.19.2 memcmp

The `memcmp` function compares the first *len* bytes of *buf1* and *buf2* and returns a value indicating their relationship as listed below.

int memcmp(void *****buf1,* **void *****buf2*, **int *len*);

Return Value

< 0	*buf1* less than *buf2*
= 0	*buf1* identical to *buf2*
> 0	*buf1* greater than *buf2*

Parameters

input	*buf1*	first input buffer
	buf2	second input buffer
	len	number of characters to compare

3.19.3 memcpy

The `memcpy` function copies *len* characters from the source (*src*) to the destination (*dst*). If some regions of the source area and the destination overlap, the `memcpy` function ensures that the original source bytes in the overlapping region are copied before being overwritten.

void *memcpy(void *****dst,* **void *****src*, **int *len*);

Return Value

The function returns the value of *dst*.

Parameters

input	*dst*	destination object
	src	source object
	len	number of characters to copy

3.19.4 memmove

The `memmove` function copies *len* characters from the source (*src*) to the destination (*dst*). If some regions of the source area and the destination overlap, the `memmove` function ensures that the original source bytes in the overlapping region are copied before being overwritten.

void *memmove(void *****dst,* **void *****src*, **int *len*);

Return Value

The function returns the value of *dst*.

Parameters

input	*dst*	destination object
	src	source object
	len	number of characters to copy

3.19.5 memset

The `memset` function sets the first *len* bytes of *dst* to the character *c*.

void *memset(void *_dst_, char _c_, int _len_);

Return Value

The function returns the value of *dst*

Parameters

input	*dst*	pointer to destination
	c	character to set
	len	number of characters

3.19.6 sprintf

The `sprintf` function formats and stores a series of characters and values in *buffer*. Each argument (if any) is converted and output according to the corresponding format specification in *format*. The format consists of ordinary characters. A null character is appended to the end of the characters written, but is not counted in the return value.

The format is composed of zero or more directives: ordinary wide characters and conversion specifications. Each conversion specification has the following form:

% [*flag*] [*width*] [*type*]

- flags supported:
  - - causes the result to be left justified within the field;
  - 0 for d,u,x conversions, leading zeros (following any indication of sign) are used to pad to the field width, no space padding is performed. If the 0 and - flags both appear, the 0 flag is ignored.
- width:
  - Minimum field width (positive integer). If the converted value has fewer wide characters than the field width, it is left padded with spaces(by default).
- types supported:
  - d the int argument is converted to signed decimal in the style *[-]dddd;*
  - u the unsigned int is converted into decimal notation;
  - X the unsigned int is converted into hexadecimal notation, the letters ABCDEF are used;
  - c for a character;
  - s for a `NULL` terminated string.

- All other features are not supported.
- If a conversion specification is invalid, the behavior is undefined.

int sprintf(char **buffer***, char** **format ...* **);**

Return Value

The `sprintf` function returns the number of characters stored in *buffer* not including the `NULL` character.

Parameters

input	*buffer*	storage location for output
	format	format control string

3.19.7 strcat

The `strcat` function appends *str2* to *str1*, terminates the resulting string with a null character, and returns a pointer to the concatenated string (*str1*).

char *strcat(char ***_str1_, char ***_str2_**);**

Return Value

The function strcat returns a pointer to the concatenated string.

Parameters

input	*str1*	destination string
	str2	source string

3.19.8 strchr

The `strchr` function returns a pointer to the first occurrence of *c* in *str*. The character *c* may be the null character ('\0'); the terminating null character of *str* is included in the search. The function returns `NULL` if the character is not found.

char *strchr(char **str***, char** *c***);**

Return Value

The function strchr returns a pointer to the first occurrence of *c* in *str*.

Parameters

input	*str*	source string
	c	the character to locate

3.19.9 strcmp

The `strcmp` function compares *str1* and *str2* lexicographically. The `strcmp` function operates on null-terminated strings.

int strcmp(char ****str1*, char ****str2** **);**

Return Value

< 0	*str1* less than *str2*
= 0	*str1* identical to *str2*
> 0	*str1* greater than *str2*

Parameters

input	*str1*	string to compare
	str2	string to compare

3.19.10 strcpy

The `strcpy` function copies *str2*, including the terminating null character, to the location specified by *str1*, and returns *str1*. The `strcpy` function operates on null-terminated strings.

char *strcpy(char **str1***, **char** **str2* **);**

Return Value

The `strcpy` function returns a pointer to *str1*.

Parameters

input	*str1*	destination string
	str2	source string

3.19.11 strlen

The `strlen` function returns the length, in bytes, of *str*, not including the terminating null character ('\0').

int strlen(char **str* **);**

Return Value

The `strlen` function returns the length of *str.*

Parameters

input	*str*	null terminalted string

3.19.12 strncat

The `strncat` function appends, at most, the first *len* characters of *str2* to *str1*, terminates the resulting string with a null character ('\0'), and returns a pointer to the concatenated string (*str1*). If *len* is greater than the length of *str2*, the length of *str2* is used in place of *len*.

char *strncat(char **str1***, **char** **str2***, **int** *len* **);**

Return Value

The `strncat` function returns a pointer to *str1*.

Parameters

input	*str1*	destination string
	str2	source string
	len	number of characters to append

3.19.13 strncmp

The `strncmp` function lexicographically compares, at most, the first *len* characters of *str1* and *str2* and returns a value indicating the relationship between the substrings, as listed below.

int strncmp(char **str1***, char** **str2***, int** *len* **);**

Return Value

< 0	*str1* less than *str2*
= 0	*str1* identical to *str2*
> 0	*str1* greater than *str2*

Parameters

input	*str1*	string to compare
	str2	string to compare
	len	number of characters compared

3.19.14 strncpy

The `strncpy` function copies *len* characters of *str2* to *str1* and returns *str1*. If *len* is less than the length of *str2*, a null character ('\0') is not appended automatically to the copied string. If *len* is greater than the length of *str2*, the *str1* result is padded with null characters ('\0') up to length *len*.

char *strncpy(char **str1***, char** **str2***, int** *len* **);**

Return Value

The function returns a pointer to *str1*.

Parameters

input	*str1*	destination string
	str2	source string
	len	number of characters copied

Appendix A: Exceptions and I/O Return Codes

This section includes all codes used as arguments to the exception handling function `XCP_throw` and all I/O related status codes. Status codes are normally used as returned *ior* (I/O result) values, but may also be used as throw codes.

Table 23 below shows the throw codes that are OTA specific. Table 24 is a list of the generic throw codes that may be thrown by the library functions. Table 25 gives the OTA I/O status codes. Values are given in both hex and decimal; the hex notation emphasises the OTA list organisation by device type and error category. For OTA-specific codes, the least significant byte gives the error code within the category and the next significant byte gives the category.

Decimal	Hex	Description	Defined Name
-4095	FFFFF001	Invalid record number. Record number outside the range defined for the current structure (zero to `DBS_avail()`-1).	XCP_DB_INVALID_RECORD
-4094	FFFFF002	Invalid function. Function inappropriate for this kind of database.	XCP_DB_INVALID_FUNCTION
-4093	FFFFF003	Database creation error. An NV structure could not be initialised (e.g. file marked read-only.).	XCP_DB_CREATION_ERROR
-4092	FFFFF004	Database access error. An NV structure could not be accessed (e.g. error in opening the file).	XCP_DB_ACCESS_ERROR
-4091	FFFFF005	Database close error. An NV structure could not be closed.	XCP_DB_CLOSE_ERROR
-4090	FFFFF006	Database seek error. A selected record could not be found in an NV structure.	XCP_DB_SEEK_ERROR
-4089	FFFFF007	Database read error. An NV structure could not be read.	XCP_DB_READ_ERROR
-4088	FFFFF008	Database write error. An NV structure could not be written.	XCP_DB_WRITE_ERROR
-4087	FFFFF009	No database selected. Attempt to use a database function prior to first use of `DBS_make_current()`.	XCP_DB_NOT_SELECTED
-3839	FFFFF101	Undefined TLV. Cannot find TLV tag name.	XCP_UNDEFINED_TLV
-3838	FFFFF102	TLV too large. TLV value field is longer than 252 bytes.	XCP_TLV_TOO_LARGE
-3837	FFFFF103	Duplicate TLV. A module contains a TLV definition that already exists.	XCP_DUPLICATE_TLV
-3583	FFFFF201	Cannot load module.	XCP_MOD_CANNOT_LOAD
-3582	FFFFF202	Cannot add module.	XCP_MOD_CANNOT_ADD
-3581	FFFFF203	Cannot open repository.	XCP_MOD_OPEN_ERROR
-3580	FFFFF204	Error creating repository.	XCP_MOD_CREATE_ERROR

Table 23: OTA THROW codes

Decimal	Hex	Description	Defined Name
-3579	FFFFF205	Error reading repository.	XCP_MOD_READ_ERROR
-3578	FFFFF206	Error closing repository.	XCP_MOD_CLOSE_ERROR
-3577	FFFFF207	Invalid module length.	XCP_MOD_INVALID_LENGTH
-3576	FFFFF208	Deletion of module not allowed.	XCP_MOD_DELETE_NOT_ALLOWED
-3575	FFFFF209	Bad card module.	XCP_MOD_BAD_CARD_MODULE
-3574	FFFFF20A	Invalid token.	XCP_MOD_INVALID_TOKEN
-3573	FFFFF20B	Not a valid module.	XCP_MOD_INVALID_MDF
-3572	FFFFF20C	Relocation error.	XCP_ERRCLASS_MEM
-3571	FFFFF20D	Error loading module.	XCP_OUT_OF_MEMORY
-3570	FFFFF20E	Invalid override.	XCP_MOD_INV_OVERRIDE
-3569	FFFFF20F	Module not in repository.	XCP_MOD_NOT_AVAILABLE
-3567	FFFFF211	Invalid character in MID.	XCP_MOD_INVALID_MID_CHAR
-3566	FFFFF212	Invalid MID string format.	XCP_MOD_INVALID_MID
-3565	FFFFF213	Cannot execute module.	XCP_MOD_CANT_EXECUTE
-3564	FFFFF214	Module in use.	XCP_MOD_IN_USE
-3327	FFFFF301	Invalid socket.	XCP_SOC_INV_SOCKET
-3071	FFFFF401	Out of memory. Request to allocate memory cannot be satisfied.	XCP_OUT_OF_MEMORY
-3070	FFFFF402	Memory release error.	XCP_RELEASE_ERROR
-3069	FFFFF403	Memory resize error.	XCP_RESIZE_ERROR
-3068	FFFFF404	Bad memory handle.	XCP_BAD_HANDLE
-3067	FFFFF405	Attempt to write outside allocated memory.	XCP_WRITE_OUT_OF_RANGE
-3066	FFFFF406	Frame stack error. Frame of the requested size could not be built.	XCP_FRAME_STACK_ERROR
-511	FFFFFE01	Illegal operation. Illegal instruction byte code.	XCP_ILLOP
-510	FFFFFE02	Language file full. Insufficient space in terminal language tables.	XCP_TOO_MANY_LANGUAGES
-509	FFFFFE03	Out of context. Attempt to use a token out of proper sequence.	XCP_OUT_OF_CONTEXT
-508	FFFFFE04	Feature not implemented. Reference to a feature or device not supported in this kernel.	XCP_UNIMPLEMENTED
-507	FFFFFE05	String too large. String size is greater than 255 bytes.	XCP_STRING_TOO_LARGE
-506	FFFFFE06	Digit too large. Digit not within number conversion base.	XCP_DIGIT_TOO_LARGE

Table 23: OTA THROW codes *(continued)*

Decimal	Hex	Description	Defined Name
-53	FFFFFFCB	Exception stack overflow.	XCP_STACK_OVERFLOW
-24	FFFFFFE8	Invalid numeric argument.	XCP_INVALID_NUMERIC_ARG
-23	FFFFFFE9	Address alignment exception.	XCP_ADDRESS_ALIGNMENT
-21	FFFFFFEB	Unsupported operation.	XCP_UNSUPPORTED_OPERATION
-17	FFFFFFEF	Pictured numeric output string overflow.	XCP_OUTPUT_STRING_OVERFLOW
-12	FFFFFFF4	Argument type mismatch.	XCP_ARG_TYPE_MISMATCH
-11	FFFFFFF5	Result out of range.	XCP_OUT_OF_RANGE
-10	FFFFFFF6	Division by zero.	XCP_ZERO_DIVIDE
-9	FFFFFFF7	Invalid memory address.	XCP_INVALID_ADDRESS
-7	FFFFFFF9	Do loops nested too deeply during execution.	XCP_TOO_MANY_DO_LOOPS
-6	FFFFFFFA	Return stack underflow.	XCP_RETURN_STACK_UNDERFLOW
-5	FFFFFFFB	Return stack overflow.	XCP_RETURN_STACK_OVERFLOW
-4	FFFFFFFC	Data stack underflow.	XCP_DATA_STACK_UNDERFLOW
-3	FFFFFFFD	Data stack overflow.	XCP_DATA_STACK_OVERFLOW

Table 24: Generic THROW codes

Decimal	Hex	Description	Defined Name
0	0	Successful operation.	DEV_OPERATION_SUCCESSFUL
-32767	FFFF8001	Device valid but currently not responding.	XCP_DEV_NOT_PRESENT
-32766	FFFF8002	Time-out.	XCP_DEV_TIMEOUT
-32765	FFFF8003	Operation cancelled by user.	XCP_DEV_CANCELLED
-32764	FFFF8004	Device Error.	XCP_DEV_ERROR
-32763	FFFF8005	Unsupported device.	XCP_DEV_UNSUPPORTED
-32762	FFFF8006	Device must be initialised.	XCP_DEV_NOT_INITIALISED
-32761	FFFF8007	Device busy.	XCP_DEV_BUSY
-32760	FFFF8008	Insufficient resources.	XCP_INSUFFICIENT_RESOURCES
-32759	FFFF8009	Device must be opened.	XCP_DEV_MUST_BE_OPENED
-32758	FFFF800A	Device already opened.	XCP_DEV_ALREADY_OPENED
-32757	FFFF800B	Device can not be opened.	XCP_DEV_CANNOT_BE_OPENED
-32239	FFFF8211	Outside border.	XCP_DSP_OUTSIDE_BORDER
-31983	FFFF8311	Printer Off-line.	XCP_PRN_OFFLINE
-31982	FFFF8312	Printer out of paper.	XCP_PRN_OUT_OF_PAPER
-31981	FFFF8313	Printer has asserted error signal.	XCP_PRN_ERROR_ASSERTED
-31980	FFFF8314	Printer does not appear to be connected.	XCP_PRN_NOT_CONNECTED
-31727	FFFF8411	No connection (for on-line commands).	XCP_MDM_NO_CONNECTION
-31726	FFFF8412	Invalid serial or modem parameters.	XCP_MDM_INVALID_PARAM
-31725	FFFF8413	No carrier.	XCP_MDM_NO_CARRIER
-31724	FFFF8414	No answer.	XCP_MDM_NO_ANSWER

Table 25: OTA I/O return codes

Decimal	Hex	Description	Defined Name
-31723	FFFF8415	Busy.	XCP_MDM_BUSY
-31722	FFFF8416	No dial tone.	XCP_MDM_NO_DIAL_TONE
-31721	FFFF8417	Modem already connected.	XCP_MDM_ALREADY_CONNECTED
-31720	FFFF8418	Connection in progress.	XCP_MDM_CONNECTION_IN_PROG
-31471	FFFF8511	Mute card (no answer).	XCP_ICC_MUTE
-31470	FFFF8512	No card in reader.	XCP_ICC_NOT_PRESENT
-31469	FFFF8513	Transmission error.	XCP_ICC_COMM_ERROR
-31468	FFFF8514	Card buffer overflow.	XCP_ICC_BUFFER_OVERFLOW
-31467	FFFF8515	Protocol error.	XCP_ICC_PROTOCOL_ERROR
-31466	FFFF8516	Response has no switches.	XCP_ICC_NO_SWITCHES
-31465	FFFF8517	Invalid buffer.	XCP_ICC_INVALID_BUFFER
-31464	FFFF8518	Other card error.	XCP_ICC_OTHER_ERROR
-31463	FFFF8519	Card partially in reader.	XCP_ICC_PARTIALLY_IN_READER
-31215	FFFF8611	Transmission error between reader and magstripe.	XCP_MAG_COMM_ERROR
-31214	FFFF8612	Output buffer overflow.	XCP_MAG_BUFF_OVERFLOW
-31213	FFFF8613	Write operation failed.	XCP_MAG_WRITE_ERROR
-30455	FFFF8909	Vending terminated.	XCP_VEND_TERMINATE

Table 25: OTA I/O return codes *(continued)*

Appendix B: Device Control

This section provides reference information for generic device control in OTA, as well as descriptions of some specific devices commonly attached to payment terminals.

B.1 Device References

Each device, including those whose lower-level operation is hidden by the Virtual Machine behind device-specific functions, shall be assigned an 8-bit device type (used to categorise result codes) and a unique device number. .

The standard OTA device number assignments are shown in Table 26.

Device Number	Description	Device Name
-1	Debug device	DEV_DEBUG
0	Primary keyboard	DEV_KEYBOARD
1	Primary display	DEV_DISPLAY
2	System printer	DEV_SYSTEM_PRINTER
3	Ticket or receipt printer	DEV_TICKET_PRINTER
4	Modem	DEV_MODEM
5	ICC card reader 1	DEV_ICC_1
6	ICC card reader 2	DEV_ICC_2
7	Magnetic stripe device	DEV_MAGSTRIPE
8	Secondary keyboard	DEV_KEYBOARD_2
9	Secondary display	DEV_DISPLAY_2
10	Secondary serial port	DEV_SERIAL_2
11	First parallel port	DEV_PARALLEL_1
12	Second parallel port	DEV_PARALLEL_2
13	Power management device	DEV_POWER
14	Vending machine device	DEV_VEND
15-31	RFU	
32-47	Serial Ports	DEV_SERIAL_3 ... DEV_SERIAL_18
48-63	Modems	DEV_MODEM_2 ... DEV_MODEM_17
64-79	Printers	DEV_PRINTER_3 ... DEV_PRINTER_18
80-...	RFU	

Table 26: Device code assignments

B.2 Debug Device

A “debug device” is the generic name for a programmer’s interactive interface to a terminal under development. The debug device is typically implemented using a host computer’s keyboard and display, communicating with the terminal through an interactive development link. Table 27 gives the *ior*s appropriate for a debug device

Status Decimal	Status Hex	Description	Name
0	0	Successful operation.	DEV_OPERATION_SUCCESSFUL
-32767	FFFF8001	Device valid but currently not responding.	XCP_DEV_NOT_PRESENT
-32766	FFFF8002	Time-out.	XCP_DEV_TIMEOUT
-32765	FFFF8003	Operation cancelled by user.	XCP_DEV_CANCELLED
-32764	FFFF8004	Device Error.	XCP_DEV_ERROR
-32763	FFFF8005	Unsupported device.	XCP_DEV_UNSUPPORTED
-32762	FFFF8006	Device must be initialised.	XCP_DEV_NOT_INITIALISED
-32761	FFFF8007	Device busy.	XCP_DEV_BUSY
-32760	FFFF8008	Insufficient resources.	XCP_INSUFFICIENT_RESOURCES
-32759	FFFF8009	Device must be opened.	XCP_DEV_MUST_BE_OPENED
-32758	FFFF800A	Device already opened.	XCP_DEV_ALREADY_OPENED
-32757	FFFF800B	Device can not be opened.	XCP_DEV_CANNOT_BE_OPENED

Table 27: Debug device I/O return codes

B.3 Keyboard Handling

The Virtual Machine returns codes for keystrokes as a set of standard values consistent across all terminals. Each returned keystroke shall be a 16-bit extended character (*echar*) value, where the more significant byte is an extension code, and the less significant byte is a key value. The key values comply with ISO 646-1983 for values in the range 0x20 - 0x7E. The standard keyboard encodings are given in Table 28 (all key values in hexadecimal). Table 29 gives the *ior*s appropriate for keyboards and Table 30 specifies the `DEV_ioctl` functions for the keyboard device.

Key character / function	Extension	Key Value
ASCII printing characters ‘ ’ (space) through ‘~’ (tilde)	00	20 .. 7E
BACKSPACE (destructive backspace)	00	08
ENTER (complete and process entry) (Green key)	00	0D
CANCEL (clear to beginning of entry) (Red key)	00	18
CLEAR (clear entry) (Yellow key)	00	7F
Function Key 1 .. Function Key 10	F0	3B .. 44
Function Key 11 .. Function Key 12	F0	85 .. 86

Table 28: Standard key mappings

Key character / function	Extension	Key Value
HOME	F0	47
END	F0	4F
Cursor Up	F0	48
Cursor Down	F0	50
Cursor Left	F0	4B
Cursor Right	F0	4D
Page Up	F0	49
Page Down	F0	51
INSERT	F0	52
DELETE	F0	53
00 (double zero key)	F0	54

Table 28: Standard key mappings *(continued)*

Status Decimal	Status Hex	Description	Name
0	0	Successful operation.	DEV_OPERATION_SUCCESSFUL
-32767	FFFF8001	Device valid but currently not responding.	XCP_DEV_NOT_PRESENT
-32766	FFFF8002	Time-out.	XCP_DEV_TIMEOUT
-32764	FFFF8004	Device Error.	XCP_DEV_ERROR
-32763	FFFF8005	Unsupported device.	XCP_DEV_UNSUPPORTED
-32762	FFFF8006	Device must be initialised.	XCP_DEV_NOT_INITIALISED
-32761	FFFF8007	Device busy.	XCP_DEV_BUSY
-32760	FFFF8008	Insufficient resources.	XCP_INSUFFICIENT_RESOURCES
-32759	FFFF8009	Device must be opened.	XCP_DEV_MUST_BE_OPENED
-32758	FFFF800A	Device already opened.	XCP_DEV_ALREADY_OPENED
-32757	FFFF800B	Device can not be opened.	XCP_DEV_CANNOT_BE_OPENED

Table 29: Keyboard I/O return codes

Function code *(fn)*	Function Name	Meaning
04	DEV_FN_TIME_OUT	Set time-out. *args* points to a cell containing the time in milliseconds before `DEV_timed_read` returns. *len* is set to to 1. The time-out value only needs to be set once and will be retained by the Virtual Machine until it is explicitly changed or the device is rebooted.

Table 30: DEV_ioctl parameters for keyboard device

B.4 Display and Printer Output

Devices that support text output (display and printer devices) provide interpretation of certain control characters appearing as output characters either individually or in output strings. The codes and the effects they produce are listed in Table 31; codes are specified in hexadecimal. Codes in the range 0x0-0x1F not appearing in this table are implementation specific.

Code	Effect
07	Emit an audible tone (if possible).
08	Destructive backspace (display only). Ior -31463 (XCP_DSP_OUTSIDE_BORDER) will be returned when this control character is written to the display any time the cursor is positioned at the beginning of a line.
0A	Carriage return and line feed. When the cursor is positioned on the last line of the display, the terminal will behave in one of the following ways : • the cursor does not move and ior -31463 (XCP_DSP_OUTSIDE_BORDER) is returned. • the cursor is positioned at the beginning of the first line. • all lines are scrolled one line up and the cursor is positioned at the beginning of the last line.
0C	Form feed. Effect a physical or visual break on printer devices; clear the screen on display devices.
0D	Carriage return and line feed.

Table 31: Control Code Interpretation

Regarding out of bounds behaviour on the display and printer device, the terminal does not inadvertently write out of bounds or hide away something the application programmer expected to be displayed or printed. Therefore, one of the following options will be implemented by the terminal when a character (that is not a control character listed in Table 31) is written to the display or printer device when the cursor is positioned at the end of a line:

- do nothing and return ior -31463 (XCP_DSP_OUTSIDE_BORDER).
- scroll the line one character to the left and add the character at the end of the line (display only).
- write the character at the beginning of the next line. When the cursor is positioned at the last line of the display, one of the following options will be implemented:
  - do nothing and return ior -31463 (XCP_DSP_OUTSIDE_BORDER).
  - write the character at the beginning of the first line.
  - scroll all lines one line up and write the character at the beginning of the last line.

Display *ior* codes are given in Table 32, and printer *ior* codes are given in Table 33. The `DEV_ioctl` arguments are specified in Table 34 and Table 35.

Status Decimal	Status Hex	Description	Name
0	0	Successful operation.	DEV_OPERATION_SUCCESSFUL
-32767	FFFF8001	Device valid but currently not responding.	XCP_DEV_NOT_PRESENT
-32764	FFFF8004	Device Error.	XCP_DEV_ERROR
-32763	FFFF8005	Unsupported device.	XCP_DEV_UNSUPPORTED
-32762	FFFF8006	Device must be initialised.	XCP_DEV_NOT_INITIALISED
-32761	FFFF8007	Device busy.	XCP_DEV_BUSY
-32760	FFFF8008	Insufficient resources.	XCP_INSUFFICIENT_RESOURCES

Table 32: Display I/O return codes

Status Decimal	Status Hex	Description	Name
-32759	FFFF8009	Device must be opened.	XCP_DEV_MUST_BE_OPENED
-32758	FFFF800A	Device already opened.	XCP_DEV_ALREADY_OPENED
-32757	FFFF800B	Device can not be opened.	XCP_DEV_CANNOT_BE_OPENED
-32239	FFFF8211	Outside border	XCP_DSP_OUTSIDE_BORDER

Table 32: Display I/O return codes *(continued)*

Status Decimal	Status Hex	Description	Name
0	0	Successful operation.	DEV_OPERATION_SUCCESSFUL
-32767	FFFF8001	Device valid but currently not responding.	XCP_DEV_NOT_PRESENT
-32766	FFFF8002	Time-out.	XCP_DEV_TIMEOUT
-32765	FFFF8003	Operation cancelled by user.	XCP_DEV_CANCELLED
-32764	FFFF8004	Device Error.	XCP_DEV_ERROR
-32763	FFFF8005	Unsupported device.	XCP_DEV_UNSUPPORTED
-32762	FFFF8006	Device must be initialised.	XCP_DEV_NOT_INITIALISED
-32761	FFFF8007	Device busy.	XCP_DEV_BUSY
-32760	FFFF8008	Insufficient resources.	XCP_INSUFFICIENT_RESOURCES
-31983	FFFF8311	Printer Off-line.	XCP_PRN_OFFLINE
-32759	FFFF8009	Device must be opened.	XCP_DEV_MUST_BE_OPENED
-32758	FFFF800A	Device already opened.	XCP_DEV_ALREADY_OPENED
-32757	FFFF800B	Device can not be opened.	XCP_DEV_CANNOT_BE_OPENED
-31982	FFFF8312	Printer out of paper.	XCP_PRN_OUT_OF_PAPER
-31981	FFFF8313	Printer has asserted error signal.	XCP_PRN_ERROR_ASSERTED
-31980	FFFF8314	Printer does not appear to be connected.	XCP_PRN_NOT_CONNECTED

Table 33: Printer I/O return codes

Function code *(fn)*	Function Name	Meaning
01	DEV_FN_DISP_CURSOR	Specify cursor behaviour. *len* is set to 1. *args* points to a cell containing a bitmap coded as follows (bit 0 is the LSB of Byte 1) : bit 0 : 0=off, 1=on bit 1 : 0=steady, 1=flash bit 2 : 0=block, 1=underscore
02	DEV_FN_DISP_BACKLIGHT	Set backlight on/off. *len* is set to 1. *args* points to a cell coded as follows : 1 cell = on/off (0=off,1=on)

Table 34: DEV_ioctl parameters for display device

Function code *(fn)*	Function Name	Meaning
03	DEV_FN_DISP_MODE	Specify display mode. *len* is set to 1. *args* points to a cell containing a bitmap coded as follows (bit 0 is the LSB of Byte 1) : bit 0 : 0=blinking off, 1=blinking on bit 1 : 0=reverse video off, 1=reverse video on bit 2 : 0=underline off, 1=underline on The display mode defined by this function is valid for all characters written to the display after this DEV_ioctl function has been executed. Other characters already displayed on the display remain unchanged.
05	DEV_FN_DISP_SIZE	Return display characteristics. *len* is set to 2. On return, *args* points to two cells to be interpreted as follows : 1 cell = width of the display (number of characters) 1 cell = height of the display (number of rows)
06	DEV_FN_BEEP_FREQ	Specify frequency of beep. *len* is set to 1. *args* points to a cell containing the frequency in Hertz of the beep when control character 07 is written to the display device.
07	DEV_FN_BEEP_DURATION	Specify duration of beep. *len* is set to 1. *args* points to a cell containing the time in milliseconds for the duration of the beep when control character 07 is written to the display device.

Table 34: DEV_ioctl parameters for display device

Function code *(fn)*	Function Na:e	Meaning
01	DEV_FN_SERIAL_PORT_INIT	Initialise serial port. *len* is set to 5. *args* points to 5 cells containing the following items : 1 cell = input baud rate 1 cell = output baud rate 1 cell = parity (0=none, 1=odd, 2=even) 1 cell = number of data bits (7 or 8) 1 cell = number of stop bits (1 or 2)
02	DEV_FN_PRINTER_FLUSH	Print whatever is already in the buffer. *len* is set to 0. args is not used by this function and may be any arbitrary non zero value.

Table 35: DEV_ioctl parameters for printer device

B.5 Serial Port Management

Control of the serial devices is provided by I/O control functions implemented through the generic `DEV_ioctl` function documented in Section 3.4.8, where *dev* is the serial device code. Other stack arguments are documented in Table 36 below. All values are in hexadecimal See Table 37 below for *ior* interpretations.

Function code *(fn)*	Function Name	Meaning
01	DEV_FN_SERIAL_PORT_INIT	Initialise serial port. *args* contains the following items: 1 cell = input baud rate 1 cell = output baud rate 1 cell = parity (0=none, 1= odd, 2=even) 1 cell = number of data bits (7 or 8) 1 cell = number of stop bits (1 or 2)
04	DEV_FN_TIME_OUT	Set port time-out. *args* points to a cell containing the time in milliseconds before `DEV_timed_read` returns. *len* is set to 1. This value only needs to be set once and will be retained by the Virtual Machine until it is explicitly changed or the device is rebooted.
05	DEV_FN_SERIAL_LINE_STATUS	Return flags indicating line status. *len* is set to 1. On input, *args* points to a cell containing the bitmap that indicates which statuses are requested to be returned. A bit set to 1 indicates that the corresponding function has to be executed. On return, *args* points to a cell containing the result to be interpreted as follows : bit 0 = true if a character is waiting to be read bit 1 = true if a character is waiting to be sent Note that a given bit position is only valid if on input the corresponding bit has been set to 1. If one of the functions requested is not implemented by the underlying Virtual Machine, -21 (`XCP_UNSUPPORTED_OPERATION`) will be returned and the complete result is invalid.

Table 36: DEV_ioctl parameters for serial port device

Status Decimal	Status Hex	Description	Name
0	0	Successful operation.	DEV_OPERATION_SUCCESSFUL
-32767	FFFF8001	Device valid but currently not responding.	XCP_DEV_NOT_PRESENT
-32766	FFFF8002	Time-out.	XCP_DEV_TIMEOUT
-32764	FFFF8004	Device Error.	XCP_DEV_ERROR
-32763	FFFF8005	Unsupported device.	XCP_DEV_UNSUPPORTED
-32762	FFFF8006	Device must be initialised.	XCP_DEV_NOT_INITIALISED
-32761	FFFF8007	Device busy.	XCP_DEV_BUSY
-32760	FFFF8008	Insufficient resources.	XCP_INSUFFICIENT_RESOURCES
-31726	FFFF8412	Invalid serial parameters.	XCP_MDM_INVALID_PARAM
-32759	FFFF8009	Device must be opened.	XCP_DEV_MUST_BE_OPENED
-32758	FFFF800A	Device already opened.	XCP_DEV_ALREADY_OPENED
-32757	FFFF800B	Device can not be opened.	XCP_DEV_CANNOT_BE_OPENED

Table 37: Serial Port I/O return codes

B.6 Modem Handling

Modem handling is provided by the functions documented in Section 3.4. The modem status (*ior*) values shown in Table 38 may be returned by these functions. Table 39 specifies the `DEV_ioctl` arguments for the modem device.

Status Decimal	Status Hex	Description	Name
0	0	Successful operation.	DEV_OPERATION_SUCCESSFUL
-32767	FFFF8001	Device valid but currently not responding.	XCP_DEV_NOT_PRESENT
-32766	FFFF8002	Time-out.	XCP_DEV_TIMEOUT
-32765	FFFF8003	Operation cancelled by user.	XCP_DEV_CANCELLED
-32764	FFFF8004	Device Error.	XCP_DEV_ERROR
-32763	FFFF8005	Unsupported device.	XCP_DEV_UNSUPPORTED
-32762	FFFF8006	Device must be initialised.	XCP_DEV_NOT_INITIALISED
-32761	FFFF8007	Device busy.	XCP_DEV_BUSY
-32760	FFFF8008	Insufficient resources.	XCP_INSUFFICIENT_RESOURCES
-31727	FFFF8411	No connection (for on-line commands).	XCP_MDM_NO_CONNECTION
-32759	FFFF8009	Device must be opened.	XCP_DEV_MUST_BE_OPENED
-32758	FFFF800A	Device already opened.	XCP_DEV_ALREADY_OPENED
-32757	FFFF800B	Device can not be opened.	XCP_DEV_CANNOT_BE_OPENED
-31726	FFFF8412	Invalid serial or modem parameters.	XCP_MDM_INVALID_PARAM
-31725	FFFF8413	No carrier.	XCP_MDM_NO_CARRIER
-31724	FFFF8414	No answer.	XCP_MDM_NO_ANSWER
-31723	FFFF8415	Busy.	XCP_MDM_BUSY
-31722	FFFF8416	No dial tone.	XCP_MDM_NO_DIAL_TONE
-31721	FFFF8417	Modem already connected.	XCP_MDM_ALREADY_CONNECTED
-31720	FFFF8418	Connection in progress.	XCP_MDM_CONNECTION_IN_PROG

Table 38: Modem I/O return codes

Function code *(fn)*	Function Name	Meaning
01	DEV_FN_SERIAL_PORT_INIT	Initialise serial port. *args* contains the following items: 1 cell = input baud rate 1 cell = output baud rate 1 cell = parity (0=none, 1= odd, 2=even) 1 cell = number of data bits (7 or 8) 1 cell = number of stop bits (1 or 2)
04	DEV_FN_TIME_OUT	Set port time-out. *args* points to a cell containing the time in milliseconds before `DEV_timed_read` returns. *len* is set to 1. This value only needs to be set once and will be retained by the Virtual Machine until it is explicitly changed or the device is rebooted.

Table 39: DEV_ioctl parameters for modem device

Function code *(fn)*	Function Name	Meaning
05	DEV_FN_SERIAL_LINE_STATUS	Return flags indicating line status. *len* is set to 1. On input, *args* points to a cell containing the bitmap that indicates which statuses are requested to be returned. A bit set to 1 indicates that the corresponding function has to be executed. On return, *args* points to a cell containing the result to be interpreted as follows : bit 0 = true if a character is waiting to be read bit 1 = true if a character is waiting to be sent Note that a given bit position is only valid if on input the corresponding bit has been set to 1. If one of the functions requested is not implemented by the underlying Virtual Machine, -21 (`XCP_UNSUPPORTED_OPERATION`) will be returned and the complete result is invalid.
06	DEV_FN_MDM_INIT	Initialise modem. *args* points to a counted string containing the modem control settings. *len* is not used by this function and may be any arbitrary value.
07	DEV_FN_MDM_INCOMING_CALL	Ask the kernel if the modem received a request for incoming call. If it did, send the counted string pointed by *args*, wait for the connection to be established and return with an ior 0. Otherwise, return immediately with an ior -31727 (`XCP_MDM_NO_CONNECTION`). *len* is not used by this function and may be any arbitrary value.

Table 39: DEV_ioctl parameters for modem device *(continued)*

B.7 ICC Card Handling

ICC card handling is provided by a set of dedicated functions documented in Section 3.6. Valid status codes returned in *ior* (I/O result) are given in Table 40 below. Table 41 specifies the `DEV_ioctl` arguments for the ICC Card Reader device.

Status Decimal	Status Hex	Description	Name
0	0	Successful operation.	DEV_OPERATION_SUCCESSFUL
-32767	FFFF8001	Device valid but currently not responding.	XCP_DEV_NOT_PRESENT
-32766	FFFF8002	Time-out.	XCP_DEV_TIMEOUT
-32765	FFFF8003	Operation cancelled by user.	XCP_DEV_CANCELLED
-32764	FFFF8004	Device Error.	XCP_DEV_ERROR
-32763	FFFF8005	Unsupported device.	XCP_DEV_UNSUPPORTED
-32762	FFFF8006	Device must be initialised.	XCP_DEV_NOT_INITIALISED
-32761	FFFF8007	Device busy.	XCP_DEV_BUSY
-32760	FFFF8008	Insufficient resources.	XCP_INSUFFICIENT_RESOURCES
-31471	FFFF8511	Mute card (no answer).	XCP_ICC_MUTE
-32759	FFFF8009	Device must be opened.	XCP_DEV_MUST_BE_OPENED
-32758	FFFF800A	Device already opened.	XCP_DEV_ALREADY_OPENED

Table 40: ICC reader I/O return codes

Status Decimal	Status Hex	Description	Name
-32757	FFFF800B	Device can not be opened.	XCP_DEV_CANNOT_BE_OPENED
-31470	FFFF8512	No card in reader.	XCP_ICC_NOT_PRESENT
-31469	FFFF8513	Transmission error.	XCP_ICC_COMM_ERROR
-31468	FFFF8514	Card buffer overflow.	XCP_ICC_BUFFER_OVERFLOW
-31467	FFFF8515	Protocol error.	XCP_ICC_PROTOCOL_ERROR
-31466	FFFF8516	Response has no switches.	XCP_ICC_NO_SWITCHES
-31465	FFFF8517	Invalid buffer.	XCP_ICC_INVALID_BUFFER
-31464	FFFF8518	Other card error.	XCP_ICC_OTHER_ERROR
-31463	FFFF8519	Card partially in reader	XCP_ICC_PARTIALLY_IN_READER

Table 40: ICC reader I/O return codes *(continued)*

Function code *(fn)*	Function Name	Meaning
01	DEV_FN_ICC_SAM_SLOT	Assign *dev* (ICC card reader 1 or ICC card reader 2) to the requested card slot. *args* points to a cell containing the slot number to select. *len* is set to 1. Slots count from 0. Ior -21 (`XCP_UNSUPPORTED_OPERATION`) will be returned if the requested slot is not supported by the reader in question. All single slot readers return 0 (Success) for slot 0 and return -21 (`XCP_UNSUPPORTED_OPERATION`) for anything else.
04	DEV_FN_TIME_OUT	Set time-out. *args* points to a cell containing the time in milliseconds before `DEV_timed_read` returns. *len* is set to 1. This value only needs to be set once and will be retained by the Virtual Machine until it is explicitly changed or the device is rebooted

Table 41: DEV_ioctl parameters for ICC card reader device

B.8 Magnetic Stripe Handling

Magnetic stripe handling is provided by a set of dedicated functions documented in Section 3.7. Valid status codes returned in *ior* (I/O result) are given in Table 42. Table 43 specifies the `DEV_ioctl` parameters for magnetic stripe device.

Status Decimal	Status Hex	Description	Name
0	0	Successful operation.	DEV_OPERATION_SUCCESSFUL
-32767	FFFF8001	Device valid but currently not responding.	XCP_DEV_NOT_PRESENT
-32766	FFFF8002	Time-out.	XCP_DEV_TIMEOUT
-32765	FFFF8003	Operation cancelled by user.	XCP_DEV_CANCELLED
-32763	FFFF8005	Unsupported device.	XCP_DEV_UNSUPPORTED

Table 42: Magnetic stripe reader I/O return codes

Status Decimal	Status Hex	Description	Name
-32762	FFFF8006	Device must be initialised.	XCP_DEV_NOT_INITIALISED
-32761	FFFF8007	Device busy.	XCP_DEV_BUSY
-32760	FFFF8008	Insufficient resources.	XCP_INSUFFICIENT_RESOURCES
-31215	FFFF8611	Transmission error between reader and magstripe.	XCP_MAG_COMM_ERROR
-32759	FFFF8009	Device must be opened.	XCP_DEV_MUST_BE_OPENED
-32758	FFFF800A	Device already opened.	XCP_DEV_ALREADY_OPENED
-32757	FFFF800B	Device can not be opened.	XCP_DEV_CANNOT_BE_OPENED
-31214	FFFF8612	Output buffer overflow.	XCP_MAG_BUFF_OVERFLOW
-31213	FFFF8613	Write operation failed.	XCP_MAG_WRITE_ERROR

Table 42: Magnetic stripe reader I/O return codes *(continued)*

Function code *(fn)*	Function Name	Meaning
04	DEV_FN_TIME_OUT	Set time-out. *args* points to a cell containing the time in milliseconds before `DEV_timed_read` and `MAG_read` return. *len* is set to 1. This value only needs to be set once and will be retained by the Virtual Machine until it is explicitly changed or the device is rebooted

Table 43: DEV_ioctl parameters for Magnetic Card reader device

B.9 Power Management

Control of the power management device is provided by I/O control functions implemented through the generic `DEV_ioctl` function documented in Section 3.4.8, where *dev* is the power management device code. `DEV_ioctl` arguments are documented in Table 44 below, and return status codes in Table 45.

Function code *(fn)*	Function Name	Meaning
01	DEV_FN_PWR_CHARGE_STATE	Return charge state of battery. *len* is set to 1. On return, *args* points to a cell coded as follows : Battery charge state : 0-100, 0=uncharged, 100=fully charged.
02	DEV_FN_PWR_SOURCE	Return current power source. *len* is set to 1. On return, *args* points to a cell coded as follows : 1 = powered by battery 2 = non depleting power source
03	DEV_FN_PWR_DOWN	Power down the terminal. *len* is set to 0. *args* is not used by this function and may be any arbitrary non zero value.
04	DEV_FN_PWR_OFF_DISALLOW	Disallow terminal power-off initiated by the "off" button. *len* is set to 0. *args* is not used by this function and may be any arbitrary non zero value.

Table 44: DEV_ioctl parameters for power management device

Function code *(fn)*	Function Name	Meaning
05	DEV_FN_PWR_OFF_ALLOW	Allow terminal power-off initiated by the "off" button. *len* is set to 0. *args* is not used by this function and may be any arbitrary non zero value.
06	DEV_FN_PWR_INQUIRE	Inquire why power has been supplied to the terminal. *len* is set to 1. On return, *args* points to a cell coded as follows : 1 = Normal Power-on 2 = Periodic wakeup 3 = Response to call

Table 44: DEV_ioctl parameters for power management device *(continued)*

Power management return status codes are given in Table 45.

Status Decimal	Status Hex	Description	Name
-32767	FFFF8001	Device valid but currently not responding.	XCP_DEV_NOT_PRESENT
-32764	FFFF8004	Device Error.	XCP_DEV_ERROR
-32763	FFFF8005	Unsupported device.	XCP_DEV_UNSUPPORTED
-32761	FFFF8007	Device busy.	XCP_DEV_NOT_INITIALISED
-32760	FFFF8008	Insufficient resources.	XCP_DEV_BUSY
-32759	FFFF8009	Device must be opened.	XCP_DEV_MUST_BE_OPENED
-32758	FFFF800A	Device already opened.	XCP_DEV_ALREADY_OPENED
-32757	FFFF800B	Device can not be opened.	XCP_DEV_CANNOT_BE_OPENED

Table 45: Power management I/O return codes

B.10 Vending Machine Control

Control of the vending machine device is provided by I/O control functions implemented through the generic `DEV_ioctl` function documented in Section 3.4.8, where *dev* is the vending machine device code, *num* is always zero, and *addr* is used to pass vending amount on certain calls. Other stack arguments are documented in Table 46 below, and valid *iors* are in Table 47 below.

Function code *(fn)*	Function Name	Meaning
01	DEV_FN_VEND_BEGIN	Begin vending session. Vending amount returned in *args*.
02	DEV_FN_VEND_CONTINUE	Continue vending session. Next vending amount returned in *args*.
03	DEV_FN_VEND_STATUS	Get vending session status.
04	DEV_FN_VEND_APPROVAL	Get vending controller transaction approval.
05	DEV_FN_VEND_DENIAL	Get vending controller transaction denial.

Table 46: DEV_ioctl parameters for vending machine device

Status Decimal	Status Hex	Description	Name
0	0	Successful operation.	DEV_OPERATION_SUCCESSFUL
-32767	FFFF8001	Device valid but currently not responding.	XCP_DEV_NOT_PRESENT
-32766	FFFF8002	Time-out.	XCP_DEV_TIMEOUT
-32765	FFFF8003	Operation cancelled by user.	XCP_DEV_CANCELLED
-32764	FFFF8004	Device Error.	XCP_DEV_ERROR
-32763	FFFF8005	Unsupported device.	XCP_DEV_UNSUPPORTED
-32762	FFFF8006	Device must be initialised.	XCP_DEV_NOT_INITIALISED
-32761	FFFF8007	Device busy.	XCP_DEV_BUSY
-32760	FFFF8008	Insufficient resources.	XCP_INSUFFICIENT_RESOURCES
-32759	FFFF8009	Device must be opened.	XCP_DEV_MUST_BE_OPENED
-32758	FFFF800A	Device already opened.	XCP_DEV_ALREADY_OPENED
-32757	FFFF800B	Device can not be opened.	XCP_DEV_CANNOT_BE_OPENED
-30455	FFFF8909	Vending terminated.	XCP_VEND_TERMINATE

Table 47: Vending machine I/O return codes

Appendix C: Operating System Calls

This section provides reference information for generic Operating System functions (`OSS_call`) in OTA. Note that not all functions have to be implemented by the Virtual Machine. However, if a function is implemented, its interface should follow the description herein, as far as the particular hardware configuration permits.

Function code *(fn)*	**Meaning**
01	Initiate terminal dependent procedure to replace the kernel. *len* is set to 0. *args* is not used by this function and may be any arbitrary non zero value.
02	Return serial number of the terminal. On return, *args* points to a counted string containing the serial number of the terminal. *len* is not used by this function and may be any arbitrary value.

Table 48: OSS_call functions

Appendix D: Alphabetical List of OTA Functions.