MITRAC CC CSS4 API Reference Manual

Contents

1 Introduction

This document lists all functions and constraints in the CSS API. The API is the interface between the real-time operating system and the application. The functions and constants are listed in service order in CSS Common API functions. There are some additional functions that are only relevant for EOS. These functions are described in CSS EOS API functions. The PIL API is described in Processor Interface Library.
This document has been written for CSS product version 4 and higher.

1.1 Purpose

The objectives of the CSS API Manual are to help in development of:

- C function applications
- CSS
- Mitrac CC Tools
- MTPE function blocks

1.2 Scope

This comprehensive document makes it unnecessary to screen for other documents except those explicitly mentioned under references in chapter References / Related Documents.

1.3 Intended Audience

This document is intended for software engineers as a reference manual. It describes how to use the Common API of CSS RTS.

1.4 Revision History

The document revision and history is presented in the chapters Revision History and Active Sheet Record

1.5 Reference Documents

[UG]	3EST000231-6857, MITRAC CC CSS4 User's Guide
[MON]	3EST000232-1881, MITRAC CC CSS4 Monitor Manual
[CSSCRM]	3EST000231-6858 MITRAC CC CSS4 Configuration Reference Manual
[NVRAM]	3EGM081110D0013, MITRAC TCMS NVRAM Handler User Manual
[TCNPD]	3EGM081140D9020, TCN Application Interface Process Data, User's Manual
[TCNMD]	3EGM081140D9021, TCN Application Interface Message Data, User's Manual
[TCNBA]	3EGM081140D9022, TCN Application Interface MVB Busadministrator, User's Manual
[MFUUM]	3EH-300300-0573, MCG Framework File Upload Library User's Manual
[MCGEDSICD]	3EH-300300-0103, TWCS MCG Framework - EDS Interface Control Document
[TRDPUM]	3EGM019001-1013, TRDP SDK User and Integration Manual
[CCUAG]	3EST000217-6940, TCMS CCU Platform Products Application Guide

1.6 Definitions and Acronyms

A list of acronyms and terms used in this document and their definitions is shown below.

Definition/ Acronym/ Abbreviation	Definition
API	Application Programming Interface
ANSI	American National Standards Institute
CC2	Second generation of controllers in the Mitrac CC product family (DCU2, and future platforms)
CSS	Product name, Common Software Structure
CPU	Central Processing Unit
CU	Control Unit
DC	Device Configuration
DLU	Downloadable Unit
EEPROM	Electrically Erasable Programmable Read Only Memory
EOS	3rd Generation Drive Control Units. EOS is a product name, not an abbreviation
FTP	File Transfer Protocol
GW	Gateway
HW	Hardware
IP	Internet Protocol
ISR	Interrupt Sub Routine
LED	Light Emitting Diode
Mitrac CC	Name of PPCs product portfolio for communication and control products
MT	Memory Test
MVB	Multifunctional Vehicle Bus
NVRAM	Non Volatile Memory
OS	Operating System
PAR	External Parameter Service
PIL	Processor Interface Library
RAM	Random Access Memory
RTS	Run-time System
SW	Software
TCB	Task Control Block
TCN	Train Communication Network, also maintenance/development project
TCP	Transmission Control Protocol
TRDP	Train Real-time Data Protocol
VCU-C2	Mitrac CC Vehicle Control Unit - Compact 2
WTB	Wire Train Bus

Table 1: Definitions and acronyms

1.7 Conventions

Each CSS Service is described briefly in the entry to each CSS Service. The CSS API functions appear under the CSS Service Group where they belong. Inside a CSS Service Group the functions are included in alphabetical order.

1.7.1 CSS API functions

The following convention is used to describe the CSS API functions:

- ---
  A solid line indicating that the description of the next API function begins here
- The name of the API function in **Bold Face**
- The API function prototype
- A description of the API function
  - First a description of the function

– Last an explanation of the return value if any

---

foo( )

Synopsis

```
int foo(
    int  parameter1,            /* parameter1 description */
    int  parameter2,            /* parameter2 description */
    int  parameter3)            /* parameter3 description */
```

Description

Here function foo( ) is described.

. . .

Returns something on success, otherwise something else.

1.8 Data types

The CSS API functions (except the ANSI functions) are declared using CSS data types.
The ANSI functions are declared according to the ANSI-C standard.

In addition, to support using code written outside of TCMS, CSS also declares common C99 types. The symbol O_C99_TYPES is defined to allow easy porting of external include files. It signals that the C99 types are declared by CSS.

The following table shows the relation between CSS types and ANSI types in the current CSS implementation.

CSS	ANSI	Description
VOID	void	Empty
BOOL	int	32-bit signed value
STATUS	int	32-bit signed value
ARGINT	int	32-bit signed value
INT8	char	8-bit signed value
SINT8	char	8-bit signed value
int8_t	char	8-bit signed value
INT16	short	16-bit signed value
int16_t	short	16-bit signed value
INT32	int	32-bit signed value
int32_t	long	32-bit signed value
int64_t	long long	64-bit signed value
BYTE	unsigned char	8-bit unsigned value
CHAR	char	8-bit signed value
UCHAR	unsigned char	8-bit unsigned value
UINT8	unsigned char	8-bit unsigned value
uint8_t	unsigned char	8-bit unsigned value
WORD	unsigned short	16-bit unsigned value
USHORT	unsigned short	16-bit unsigned value
UINT16	unsigned short	16-bit unsigned value
uint16_t	unsigned short	16-bit unsigned value
DWORD	unsigned int	32-bit unsigned value
UINT	unsigned int	32-bit unsigned value
UINT32	unsigned int	32-bit unsigned value
ULONG	unsigned long	32-bit unsigned long value
uint32_t	unsigned long	32-bit unsigned value
FLOAT32	float	32-bit floating point value
uint64_t	unsigned long long	64-bit unsigned value
FLOAT64	double	64-bit floating point value
FUNCPTR	int (*FUNCPTR)()	Pointer to function returning int
VOIDFUNCPTR	void (*VOIDFUNCPTR)()	Pointer to function returning void
DBLFUNCPTR	double (*DBLFUNCPTR)()	Pointer to function returning double
FLTFUNCPTR	float (*FLTFUNCPTR)()	Pointer to function returning float

Table 2: CSS Data Types

2 CSS Common API functions

2.1 AE - Application Event Log Service

Overview:

CSS provides an event log for applications to store error conditions. The application event log is stored in NVRAM memory, using two circular buffers of approx. 13 entries each. When a buffer becomes full the oldest entry will be over written. Entries are also stored to files as they arrive.

Two generic strategies are defined for AE-Log messages:

AE_WARNING	used for non-critical messages
AE_ERROR	used for critical messages

Log Traversal:

The functions ae_get_newest_index and ae_get_entry are the preferred API to be used for traversing a specific strategy list. Function ae_get_entry retrieves the full message from the log.

The following functions are retained for legacy purposes. They are depreacted and are not recommended for new applications:

The functions ae_last_struct_get and ae_indx_struct_get can be used to traverse a specific strategy list.
The functions ae_log_last_get and ae_log_prev_get can be used to traverse both strategy lists (starting with the most recent log entry).

Log Traversal Caveats:

Due to limitations in the legacy API functions only one AE list traversal may be active at any time. API functions ae_get_newest_index, ae_last_struct_get and ae_log_last_get all start a new traversal.
Only those entries stored in the NVRAM can be traversed using these three function groups.
There exists no API function for traversing the contents of the files used for long-term storage.

The CallBack Mechanism:

Applications may register a function that CSS calls when a new entry is inserted into the log. The call-back is called from an internal CSS task which runs at low priority and thus does not disturb normal operation.

The call-back must be registered by an init task in order to make sure that all entries are reported to the call-back function, even after a restart of the device.

If the internal NVRAM buffer overruns before the entry is reported to the application then the entry will not be reported and no report of the lost entry is provided.

The preferred API function to register a call-back function is ae_callback_add.
API function ae_callback_reg is retained for legacy applications. It is not recommended for new applications.
A call-back can be removed by calling the ae_callback_del function.

---

ae_callback_add( )

Synopsis

```
int ae_callback_add(
    LOG_CB_FUNC_PTR func,                    /* In: Function pointer */
    int             *p_cb_id)                /* Out: Id of the call-back */
```

Description

This function registers a call-back function to be called when a new message is logged.

The call-back function, takes one argument: a pointer to a struct of type LOG_ENTRY.
In this struct the AE-Log message is passed to the application through the call-back call.

Following the defines and typedef used in the call-back function:

```
  #define LOG_MESSAGE_LENGTH  250
```

```
typedef struct STR_LOG_DATA
{
    UINT32              seconds;
    INT16               strategy;
    UINT32              id;
    char                message[LOG_MESSAGE_LENGTH];
} LOG_ENTRY;
```

seconds	the time-stamp for the message
strategy	the type of message, AE_ERROR or AE_WARNING
id	the identity for the message
message	the buffer where the message is put

An identity for the registred call-back function is returned in cb_id, to be used when deleting it.

Returns OK on success or ERROR if failed to register the call-back function.

---

ae_callback_del( )

Synopsis

```
int ae_callback_del(
    int cb_id)                                  /* In: Call-back register id */
```

Description

Delete the registered call-back function. The call-back function is selected by the parameter id.

Returns OK on success or ERROR if failed to delete the call-back function.

---

ae_callback_reg( )

Synopsis

```
int ae_callback_reg(
    AEFUNCPTR  func,                            /* In: Function pointer */
    int       *p_cb_id)                         /* Out: Id of the call-back */
```

Description

Note: This function is deprecated.
Use ae_callback_add instead in new applications.

---

ae_clear( )

Synopsis

```
void ae_clear(
    void)
```

Description

This function can be used by the application to clear the AE-Log.

The function erases all the AE-Log entries.

---

ae_get_entry( )

Synopsis

```
int ae_get_entry(
    INT16        strategy,                  /* In:    AE_ERROR or AE_WARNING */
    int         *p_index,                   /* InOut: Entry index */
    LOG_ENTRY   *p_ae_data )                /* Out:   The log entry */
```

Description

This function reads the AE-Log entry indicated by *p_index. The message is stored in the buffer ae_data.

The p_index argument is the requesting index whose entry should be returned in *p_ae_data. Upon exit the value has been modified to indicate the next, older, entry to get.

The type of message of interest is defined in strategy, AE_WARNING or AE_ERROR.

Returns OK on success or ERROR if no message is available or any parameter in the call is unvalid.

---

ae_get_newest_index( )

Synopsis

```
int ae_get_newest_index(
    INT16        strategy,                  /* In:  AE_ERROR or AE_WARNING */
    int         *p_index )                  /* Out: Index to newest entry */
```

Description

Sets index to the newest logged entry in the AE-Log.

The type of message of interest is specified in strategy: AE_WARNING or AE_ERROR.

Upon exit *p_index holds the index for the newest stored entry which can be used as input to ae_get_entry.

Returns OK on success or ERROR if the last message could not be read.

---

ae_indx_struct_get( )

Synopsis

```
int ae_indx_struct_get(
    int          idx,           /* In:  Requesting index */
    int         *p_next_idx,    /* Out: Next Index  */
    int          strategy,      /* In:  Strategy of message (AE_ERROR or AE_WARNING) */
    AE_LOG_DATA *p_ae_data)     /* Out: String contains message log */
```

Description

Note: This function is deprecated.
New applications should use ae_get_newest_index/ae_get_entry instead.
This function is added for backwards compatibility only. Do NOT use this function for new applications.

This function reads the AE-Log message indicated by indx. The message is stored in the buffer ae_msg.

The idx parameter is the requesting index that should be returned as ae_msg. Range 0..(max-1). The function ae_last_struct_get can be used to get the index to the oldest structure.

The index for the next (earlier) message, if any, is returned at p_next_idx.

The type of message of interest is defined in strategy, AE_WARNING or AE_ERROR.

Returns OK on success or ERROR if no message is available or any parameter in the call is unvalid.

---

ae_info( )

Synopsis

```
void ae_info(
    INT16* p_entry_count,       /* Out: The number of entries */
    INT16* p_current_index      /* Out: Next free index */
    )
```

Description

This function prints the internal log header to standard output.

Sets the output arguments to zero.

ae_last_struct_get( )

Synopsis

```
int ae_last_struct_get(
    int           strategy,      /* In: AE_ERROR or AE_WARNING */
    AE_LOG_DATA *p_ae_data,      /* Out: String which contains the log entry */
    int          *p_idx)         /* Out: Entry index */
```

Description

Note: This function is deprecated.
New applications should use ae_get_newest_index/ae_get_entry instead.
This function is added for backwards compatibility only. Do NOT use this function for new applications.

Gets the last logged message from the AE-Log.
The message is returned in the buffer at p_ae_msg.

The type of message of interest is specified in buff_type: AE_WARNING or AE_ERROR.

Upon exit p_idx holds the index for the stored message. The value of which can be used as input to ae_indx_struct_get.

Returns OK on success or ERROR if the last message could not be read.

ae_log_last_get( )

Synopsis

```
INT16 ae_log_last_get(
    char  *log_entry,            /* Out: The AE-Log message */
    INT16 *p_idx)                /* Out: The index of the message */
```

Description

Note: This function is deprecated.
New applications should use ae_get_newest_index/ae_get_entry instead.
This function is added for backwards compatibility only. Do NOT use this function for new applications.

This function gets the latest logged message as a string in the buffer log_entry.

Upon exit p_idx holds the index for the stored message.

Returns OK on success otherwise ERROR.

ae_log_prev_get( )

Synopsis

```
int ae_log_prev_get(
    char  *p_log_entry,          /* Out:   The AE-Log message */
    INT16 *p_idx)                /* InOut: The index of the message */
```

Description

Note: This function is deprecated.
New applications should use ae_get_newest_index/ae_get_entry instead.
This function is added for backwards compatibility only. Do NOT use this function for new applications.

This function gets the previous AE-Log message in the buffer p_log_entry.

Upon entry p_idx hold the index for the current message. Upon exit p_idx holds the index for the returned message.

Returns OK on success otherwise ERROR.

ae_put( )

Synopsis

```
void ae_put(
    INT16        strategy,      /* In: Strategy AE_ERROR or AE_WARNING */
    UINT32       id,            /* In: A unique id of the message */
    const char  *fmt,           /* In: The message format */
    ... )                       /* In: Additional arguments */
```

Description

This function writes a messsage into the AE-Log.

The message can be a format message string with a list of arguments (...).

The user should identify the creator of the message by assigning different ids to different applications or modules.

---

ae_write( )

Synopsis

```
int ae_write(
    const char* p_file_id,      /* In: Pointer to FILE NAME */
    UINT16      line_nr,        /* In: Line number of the logged event */
    const char* fmt,            /* In: The formated text string */
    ... )                       /* In: Additional arguments */
```

Description

Note: This function is deprecated.
New applications should use ae_put instead.
This function is added for backwards compatibility only. Do NOT use this function for new applications.

2.2 ANSI - ANSI Service

The following standard funtions are available, for description refer to:

INTERNATIONAL STANDARD ISO9899 2'nd edition 1999-12-01 Programming Language - C.

ANSI ctype	ANSI stdlib	ANSI string	ANSI stdio	POSIX	De Facto Standard
isalnum( )	abs( )	memchr( )	fflush( )	strtok_r( )	void bcopy(const char* source, char* dest, int nbytes)
isalpha( )	atoi( )	memcmp( )	fgetc( )		void bzero(char* buffer, int nbytes)
iscntrl( )	atol( )	memcpy( )	fgets( )		
isdigit( )	bsearch( )	memmove( )	fprintf( )		
isgraph( )	labs( )	memset( )	fputc( )		
islower( )	qsort( )	strcat( )	fputs( )		
isprint( )	rand( )	strchr( )	fread( )		
ispunct( )	srand( )	strcmp( )	fscanf( )		
isspace( )	strtol( )	strcpy( )	fseek( )		
isupper( )	strtoul( )	strcspn( )	ftell( )		
isxdigit( )		strlen( )	fwrite( )		
tolower( )		strncat( )	getc( )		
toupper( )		strncmp( )	getchar( )		
		strncpy( )	gets( )		
		strpbrk( )	printf( )		
		strrchr( )	putc( )		
		strspn( )	putchar( )		
		strstr( )	puts( )		
		strtok( )	scanf( )		
			setbuf( )		
			setvbuf( )		
			snprintf( )		
			sprintf( )		
			sscanf( )		
			ungetc( )		

The following standard file and I/O functions are available:

POSIX dir	Standard C stdio	POSIX usrFs
closedir( )	rename( )	mkdir( )
opendir( )		
readdir( )		
stat( )		
unlink( )		

Global symbol for retrieving the latest error number.

- os_errno equivalent with errno
  This symbol holds the latest error number for the calling task. Some common error numbers are:
  - OS_ERRNO_OBJ_ID_ERROR - The object does not exist
  - OS_ERRNO_TIMEOUT - The time-out time has elapsed
  - OS_ERRNO_OBJ_UNAVAILABLE - The time-out time is set to NO_WAIT and the object can not return the asked resource/data

  For more examples about possible error numbers, refer to "error codes" in VxWorks Reference Manuals.

Types to be used in ANSI function calls:

- size_t
  typedef unsigned int

- time_t
  typedef unsigned long

Types used in struct stat:

- dev_t
  typedef unsigned long
- ino_t
  typedef unsigned long
- mode_t
  typedef int
- nlink_t
  typedef unsigned long
- uid_t
  typedef unsigned short
- gid_t
  typedef unsigned short
- blksize_t
  typedef long
- blkcnt_t
  typedef unsigned long

Defines used by file and I/O functions:

File mode (st_mode) bit masks

Define	Bit mask	Description
S_IFMT	0xf000	file type field
S_IFIFO	0x1000	fifo
S_IFCHR	0x2000	character special
S_IFDIR	0x4000	directory
S_IFBLK	0x6000	block special
S_IFREG	0x8000	regular
S_IFLNK	0xa000	symbolic link
S_IFSHM	0xb000	shared memory object
S_IFSOCK	0xc000	socket
S_ISUID	0x0800	set user id on execution
S_ISGID	0x0400	set group id on execution
S_IRUSR	0x0100	read permission, owner
S_IWUSR	0x0080	write permission, owner
S_IXUSR	0x0040	execute/search permission, owner
S_IRWXU	0x01c0	read/write/execute permission, owner
S_IRGRP	0x0020	read permission, group
S_IWGRP	0x0010	write permission, group
S_IXGRP	0x0008	execute/search permission, group
S_IRWXG	0x0038	read/write/execute permission, group
S_IROTH	0x0004	read permission, other
S_IWOTH	0x0002	write permission, other
S_IXOTH	0x0001	execute/search permission, other
S_IRWXO	0x0007	read/write/execute permission, other

Possible values for the whence argument to fseek

Define	Value	Description
SEEK_SET	0	Relative to the start of the file
SEEK_CUR	1	Relative to the current position
SEEK_END	2	Relative to the end of the file

File type test macros

Macro	-	Description
S_ISDIR(mode)	((mode & S_IFMT) == S_IFDIR)	directory
S_ISCHR(mode)	((mode & S_IFMT) == S_IFCHR)	character special
S_ISBLK(mode)	((mode & S_IFMT) == S_IFBLK)	block special
S_ISREG(mode)	((mode & S_IFMT) == S_IFREG)	regular file
S_ISFIFO(mode)	((mode & S_IFMT) == S_IFIFO)	fifo special
S_ISLNK(mode)	((mode & S_IFLNK) == S_IFLNK)	symbolic link special

Struct used in stat function:

```
struct stat
{
    dev_t       st_dev;          Device ID number
    ino_t       st_ino;          File serial number
    mode_t      st_mode;         Mode of file
    nlink_t     st_nlink;        Number of hard links to file
    uid_t       st_uid;          User ID of file
    gid_t       st_gid;          Group ID of file
    dev_t       st_rdev;         Device ID if special file
    long long   st_size;         File size in bytes
    time_t      st_atime;        Time of last access
    time_t      st_mtime;        Time of last modification
    time_t      st_ctime;        Time of last status change
    blksize_t   st_blksize;      File system block size
    blkcnt_t    st_blocks;       Number of blocks containing file
    UINT8       st_attrib;       DOSFS only - file attributes
    int         st_reserved1;    reserved for future use
    int         st_reserved2;    reserved for future use
    int         st_reserved3;    reserved for future use
    int         st_reserved4;    reserved for future use
};
```

The ANSI service provides several I/O functions for performing input and output.

Types and definitions to be used in I/O functions:

- OS_IO_FILE
  Object type needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an error indicator that records whether a read or write error has occurred, and an end-of-file indicator that records whether the end of the file has been reached.
- fpos_t
  Object type capable of recording all the information needed to specify uniquely every position within a file.

---

os_io_close( )

Synopsis

```
INT16 os_io_close(
    INT32 fd)            /* In: file descriptor value of device to close */
```

Description

This routine closes a device driver and frees the file descriptor. It calls the device driver to do the work.

Returns OK on success otherwise ERROR.

os_io_ctrl( )

Synopsis

```
INT16 os_io_ctrl(
    INT32  fd,            /* In:    file descriptor */
    INT32  function,      /* In:    function code */
    INT32  arg)           /* InOut: arbitrary argument input or output */
```

Description

This routine is called by the user to get information from the hardware or change the behaviour of the hardware.

The function code to the routine can be:

- OS_IO_BAUD_SET - Set the baud rate on the device
- OS_IO_BAUD_GET - Get the baud rate
- OS_IO_NO_OF_BYTE_UNREAD - Get the number of unread bytes in the input buffer
- OS_IO_NO_OF_BYTE_UNWRITTEN - Get the number of bytes in the output buffer
- OS_IO_WRITE_FLUSH - Discard all bytes currently in the output buffer
- OS_IO_READ_FLUSH - Discard all bytes currently in the input buffer
- OS_IO_PROTOHOOK - Adds a protocol hook function to be called for each input character
- OS_IO_OPTIONS_GET - Get the device option word
- OS_IO_OPTIONS_SET - Set the device option word to the specified argument:
  - OS_IO_ARG_ECHO - Echo on
  - OS_IO_ARG_CRMOD - CR always added to NL
  - OS_IO_ARG_XON - Xon protocol on
  - OS_IO_ARG_LINE - Line editing on
  - OS_IO_ARG_RAW - Raw mode
  - OS_IO_ARG_TERMINAL - Terminal mode
- OS_IO_HW_OPTION_SET - Change the driver's hardware options to the specified argument:
  - OS_IO_CREAD - Enable device receiver
  - OS_IO_CS7 - 7 bits
  - OS_IO_CS8 - 8 bit
  - OS_IO_STOPB - Send two stop bits (else one)
  - OS_IO_PARENB - Partity detect on (else off)
  - OS_IO_PARODD - Odd parity (else even)
  - OS_IO_RS485_FULL_DPLX - RS485 full duplex
  - OS_IO_RS485_HALF_DPLX - RS485 half duplex
  - OS_IO_DATACOM_7_N_1 - 7 bit, no parity, 1 stop bit
  - OS_IO_DATACOM_7_E_1 - 7 bit, even parity, 1 stop bit
  - OS_IO_DATACOM_7_O_1 - 7 bit, odd parity, 1 stop bit
  - OS_IO_DATACOM_8_N_1 - 8 bit, no parity, 1 stop bit
  - OS_IO_DATACOM_8_E_1 - 8 bit, even parity, 1 stop bit
  - OS_IO_DATACOM_8_O_1 - 8 bit, odd parity, 1 stop bit
  - OS_IO_DATACOM_7_N_2 - 7 bit, no parity, 2 stop bit
  - OS_IO_DATACOM_7_E_2 - 7 bit, even parity, 2 stop bit

  - OS_IO_DATACOM_7_O_2 - 7 bit, odd parity, 2 stop bit
  - OS_IO_DATACOM_8_N_2 - 8 bit, no parity, 2 stop bit
  - OS_IO_DATACOM_8_E_2 - 8 bit, even parity, 2 stop bit
  - OS_IO_DATACOM_8_O_2 - 8 bit, odd parity, 2 stop bit
  - OS_IO_RS422OFF - (for compatible reason)
  - OS_IO_RS422ON - (for compatible reason)
- OS_IO_DEVICE_DELETE - Delete the device accosiated with the file descriptor fd

Routine call parameter arg specifies arbitrary argument, depending on the function code.

Returns ERROR if device could not be found in device list or file descriptor does not exist otherwise OK.

---

os_io_fclose( )

Synopsis

```
INT16 os_io_fclose(
    OS_IO_FILE   *fp)        /* In: file pointer value of device to close */
```

Description

This routine flushes a specified stream and closes the associated file. Any unwritten buffered data is delivered to the host environment to be written to the file. Any unread buffered data is discarded. The stream is disassociated from the file. If the associated buffer was allocated automatically, it is deallocated.

Returns OK if the stream is closed successfully otherwise ERROR.

---

os_io_fopen( )

Synopsis

```
INT16 os_io_fopen(
    const char   *device_name,   /* In:  name of the device to open */
    const char   *mode,          /* In:  "r", "w" "r+" read, write or read/write */
    OS_IO_FILE   **fp)           /* Out: file pointer to the io device */
```

Description

This routine opens a device for reading and/or writing according to mode and associates a stream with it.

It puts the file pointer in fp for the opened device.

Valid device_name is:

Define	I/O Device	Description
	”/app0/”	File system
	”/data0/”	File system
	”/usb0”	File system supported on VCU-C2 devices
OS_IO_COM1	”/tyCo/0”	Monitor channel on VCU-C2
OS_IO_COM2	”/usb2ttyS/0”	Serial channel A on VCU-C2
OS_IO_COM3	”/usb2ttyS/1”	Serial channel B on VCU-C2
OS_IO_COM4	”/usb2ttyS/2”	Serial channel USB on VCU-C2
OS_IO_COMX	”/tyOutX”	Multicast virtual output channel. Output to all active RTS monitor channels. The only supported mode is write.

Example on how to open a file for reading and writing on Serial channel B, the device name OS_IO_COM3 should be used:

```
os_io_fopen ( OS_IO_COM3, "r+", &fp );
```

Serial channel A on VCU-C2 can be RS232 full duplex or RS485 half duplex. This is selected with the type of cable used and is not possible to configure. It is not possible to have both simultaneously.

Serial channel B and USB on VCU-C2 can be RS485 half or full duplex, full is default. The type of cable is specific to the selected mode. To select mode, use the defines OS_IO_RS485_FULL_DPLX or OS_IO_RS485_HALF_DPLX

Example on how to configure Serial channel B with full duplex, raw mode, 8 data bits, 1 stop bit, no parity and receiver enabled:

```
os_io_fopen ( OS_IO_COM3, "r+", &fp );

os_io_fp_to_fd ( fp, &fd );

os_io_ctrl ( fd, OS_IO_HW_OPTION_SET,
             OS_IO_CS8 | OS_IO_CREAD | OS_IO_RS485_FULL_DPLX );
```

The parameter mode points to a string beginning with one of the following sequences:

r	open text file for reading
w	truncate to zero length or create text file for writing
a	append; open or create text file for writing at end-of-file
rb	open binary file for reading
wb	truncate to zero length or create binary file for writing
ab	append; open or create binary file for writing at end-of-file
r+	open text file for update (reading and writing)
w+	truncate to zero length or create text file for update.
a+	append; open or create text file for update, writing at EOF
r+b/rb+	open binary file for update (reading and writing)
w+b/wb+	truncate to zero length or create binary file for update
a+b/ab+	append; open or create binary file for update, writing at EOF

Opening a file with read mode (r as the first character in the mode argument) fails if the file does not exist or cannot be read.

Opening a file with append mode (a as the first character in the mode argument) causes all subsequent writes to the file to be forced to the current end-of-file.

In some implementations, opening a binary file with append mode (b as the second or third character in the mode argument) may initially position the file position indicator for the stream beyond the last data written, because of null character padding. Whether append mode is supported is device-specific.

When a file is opened with update mode (+ as the second or third character in the mode argument), both input and output may be performed on the associated stream.

Output may not be directly followed by input without an intervening call to fflush( ).

Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

Note! It is not possible to open a HRFS device through os_io_fopen, e.g. os_io_fopen(”/app0/”, ”r+”), if so the routine will return ERROR. This does not concern opening files on the device - it is only a problem opening the entire device.

Returns OK on success or ERROR if the operation fails.

---

os_io_fp_to_fd( )

Synopsis

```
INT16 os_io_fp_to_fd(
    OS_IO_FILE  *fp,            /* In:  file pointer from which to read */
    INT32       *fd)            /* Out: file descriptor associated with the fp */
```

Description

This routine converts fp (file pointer) to fd (file descriptor). The file descriptor can be used by the routine os_io_ctrl to control the behaviour of the file.

Returns OK on success or ERROR if fp does not specify an open file.

---

os_io_open( )

Synopsis

```
INT16 os_io_open(
    const char   *device_name,     /* In:  name of the device to open */
    INT32        *fd)              /* Out: file descriptor value */
```

Description

This routine opens a device for reading and/or writing.

It puts the file descriptor in fd for the opened device.

Valid device_name is:

Define	I/O Device	Description
	”/app0/”	File system
	”/data0/”	File system
	”/usb0”	File system supported on VCU-C2 devices
OS_IO_COM1	”/tyCo/0”	Monitor channel on VCU-C2
OS_IO_COM2	”/usb2ttyS/0”	Serial channel A on VCU-C2
OS_IO_COM3	”/usb2ttyS/1”	Serial channel B on VCU-C2
OS_IO_COM4	”/usb2ttyS/2”	Serial channel USB on VCU-C2
OS_IO_COMX	”/tyOutX”	Multicast virtual output channel. Output to all active RTS monitor channels. The only supported mode is write.

Example on how to open a file descriptor for Serial channel B, the device name OS_IO_COM3 should be used:

```
os_io_open ( OS_IO_COM3, &fd );
```

Serial channel A on VCU-C2 can be RS232 full duplex or RS485 half duplex. This is selected with the type of cable used and is not possible to configure. It is not possible to have both simultaneously.

Serial channel B and USB on VCU-C2 can be RS485 half or full duplex, full is default. The type of cable is specific to the selected mode. To select mode, use the defines OS_IO_RS485_FULL_DPLX or OS_IO_RS485_HALF_DPLX

Example on how to configure Serial channel B with full duplex, raw mode, 8 data bits, 1 stop bit, no parity and receiver enabled:

```
os_io_open ( OS_IO_COM3, &fd );

os_io_ctrl ( fd, OS_IO_HW_OPTION_SET,
             OS_IO_CS8 | OS_IO_CREAD | OS_IO_RS485_FULL_DPLX );
```

Returns OK on success or ERROR in the following cases:

- device_name is not valid
- no more file descriptors are available
- the driver returns ERROR

---

os_io_read( )

Synopsis

```
INT16 os_io_read(
    INT32  fd,              /* In:  file descriptor from which to read */
    char   *p_buffer,       /* Out: pointer to the buffer to receive data */
    INT32  max_bytes,       /* In:  max. number of bytes to read into buffer */
    BOOL   block,           /* In:  controls the read access. If it is true the
```

```
                                    read access should wait for data to be
                                    available in the buffer */
    INT32   *no_bytes_read)   /* Out: the number of bytes the driver actually read
                                    from the device (between 0 and max_bytes) */
```

Description

This routine reads a number of bytes (less than or equal to max_bytes) from the specified file descriptor fd and places them in p_buffer. It calls the device driver to do the work.

If the block parameter is false the file descriptor will be polled and the result returned immediately. If there is no new data to be read the driver will return zero in the output variable no_bytes_read. If the block parameter is true and the device buffer is empty this routine waits until data arrives.

Returns OK on success or ERROR if the fd does not exist.

---

os_io_write( )

Synopsis

```
INT16 os_io_write(
    INT32   fd,               /* In:  file descriptor from which to write */
    char    *p_buffer,        /* Out: pointer to transmitting buffer */
    INT32   n_bytes,          /* In:  number of bytes to send from buffer */
    BOOL    block,            /* In:  controls the write access. If it is TRUE the
                                    write access waits for the io buffer to be
                                    able to swallow n_bytes of data
                                    otherwise n_bytes(OK)
                                    or zero bytes (ERROR) will be written */
    INT32   *no_bytes_write)  /* Out: number of bytes which the driver actually wrote */
```

Description

This routine writes a number of bytes to the specified fd. It calls the device driver to do the work.

If the specified fd transmit buffer is full the device returns with ERROR if the block parameter is FALSE.

If block is TRUE this routine permits the calling task to wait until there is empty space in the transmitting buffer.

Returns OK on success otherwise ERROR.

2.3 CFG - Configuration Service

The CSS Configuration Service provides an API for access to the Device and Application Configuration structures.

It is not intended that the application shall access and traverse the structures by themselves, but use the provided API to retrieve the information of interest.
The reason for that is that only CSS can decide which information is valid in the system. Information in the Device Configuration may have been overridden by some other source of information.

---

cfg_dev_ident_get( )

Synopsis

```
INT16 cfg_dev_ident_get(
    const CFG_DEV_IDENT **pp_dev_ident)       /* Out: Ptr to dev ident info */
```

Description

This function is used to get a pointer to the device identification structure.
The caller must not modify this information.

Returns OK if device identification is returned to the caller.
Returns ERROR if the argument is NULL.

---

cfg_get_appl_data( )

Synopsis

```
INT16 cfg_get_appl_data(
    const char     *appl_name,          /* In:  Application name */
    AS_APPL_DATA   **pp_appl_data,      /* Out: Ptr to application data array */
    INT32          *p_number)           /* Out: Number of array entries */
```

Description

Get the number of consecutive AS_APPL_DATA structures from the application configuration.

The first structure is pointed out by p_appl_data and the number of consecutive structures is specified in number.

Returns OK if one or more structures can be found in the application configuration or ERROR if the application can not be found.

---

cfg_get_application_date( )

Synopsis

```
INT16 cfg_get_application_date(
    UINT32   *p_date)                         /* Out: The application date */
```

Description

Get the application date as defined in the application configuration.

Returns OK if the date can be found in the application configuration or ERROR if the application can not be found.

---

cfg_get_application_name( )

Synopsis

```
INT16 cfg_get_application_name(
    char   *appl_name)                          /* Out: Application name */
```

Description

Get the name of the application associated with the calling task.

The parameter appl_name is a user-supplied buffer where the application name is to be stored.
appl_name buffer must be able to hold at least 16 characters (incl NULL).

Returns OK if the task name can be found in the device configuration otherwise ERROR.

cfg_get_data_dictionary( )

Synopsis

```
INT16 cfg_get_data_dictionary(
    const char         *appl_name,           /* In:  Application name */
    AS_DATA_DICTIONARY **pp_data_dictionary, /* Out: Ptr to the data dictionary array */
    INT32              *p_number)            /* Out: Number of array entries */
```

Description

Get the number of consecutive AS_DATA_DICTIONARY structures from the application configuration.

The first structure is pointed out by pp_data_dictionary and the number of consecutive structures is specified in number.

Returns OK if one or more structures can be found in the device configuration, otherwise ERROR.

cfg_get_monitor_channel( )

Synopsis

```
INT16 cfg_get_monitor_channel(
    char   **pp_monitor_channel)   /* Out: Ptr to name of the RTS-Mon serial port */
```

Description

This function can be used to find the name of the serial channel used for the RTS-monitor.
Caller must not modify this string!

The function will output the default monitor channel for the device if no monitor channel is specified in the device configuration.

Returns OK.

cfg_get_primary_if( )

Synopsis

```
INT16 cfg_get_primary_if(
    const CFG_IP_IF **ip_if_cfg)        /* Out: Ptr to if cfg info */
```

Description

This function is used to get a pointer to the config of the primary network interface.
The caller must not modify this information.

Returns OK if config is returned to the caller.
Returns ERROR if the argument is NULL or no config is found.

cfg_library_connect( )

Synopsis

```
INT16 cfg_library_connect(
    char         *app_name,         /* In: Name of the application */
    AS_VERSION   version,           /* In: Version of the application */
    void         **pp_ict)          /* Out: Application indirect call table */
```

Description

Connect an application library indirect call table.

Returns ERROR if the application version is not correct or the application can not be found in the device configuration, otherwise OK.

---

cfg_sw_info_get( )

Synopsis

```
INT16 cfg_sw_info_get(
    UINT16          list_index,     /* In: Item to get */
    AS_DLU_HEADER   *p_sw_info,     /* Out: A copy of the requested DLU header */
    UINT16          *p_nr_sw_info) /* Out: Total number of available SW items */
```

Description

Read software version information from the application and device configuration.

The indexing of software items is as follows:

- Device Configuration item (zero)
- Application items (zero or more, but must be zero if there is no device configuration)

p_sw_info must point to a user-supplied buffer large enough to hold a DLU header.
if p_nr_sw_info is not NULL then it must point to a user-supplied UINT16 variable.

Returns OK if DLU header could be found and copied.
Returns ERROR if:
- no device configuration exists
- list_index exceeds the number of software items available
- p_sw_info is NULL
- application is deferred and not yet loaded

2.4 CRC - Checksum Service

This service provide a rudimentary support for checksum verification of files.

Example:
If an application file named application1.out is to be verified a file named application1.out.crc containing the checksum in ascii is stored in the same directory.
A call to crc_file( ) will tell if the application1.out has the expected checksum.

---

crc_file( )

Synopsis

```
UINT16 crc_file(
    char* p_file_name)      /* In: Filename to verify */
```

Description

This function verifies the file defined by p_file_name against a checksum. A checksum file named the same as the file to verify must be stored in the same directory. The checksum file must have the extension ".crc" and contain the expected checksum. The checksum is computed using the algorithm byte addition in a buffer of type unsigned long.

Returns OK if the checksum match otherwise 0xFFFF.

2.5 DM - Dynamic Memory Service

CSS provides an interface to manage dynamic memory.
Using several partitions is useful for eliminating internal memory fragmentation. In that case create one partition for each size of used dynamic memory.
Partition 0 (zero) is always defined to be the system heap.
A maximum of 254 additional partitions may be created in the device configuration.

It is possible to allocate memory from specified partitions, and to free allocated blocks.

Note that the partition manager uses 200 bytes from the pertition for internal book-keeping.

CSS also provides an interface to allocate cache-safe buffers.

---

dm_free( )

Synopsis

```
INT16 dm_free(
    UINT8  identity,         /* In: identifier of the memory partition
                                    to add block to */
    void   *p_block)         /* In: pointer to the block of memory to free. */
```

Description

De-allocates a previously allocated block of memory (allocated using dm_malloc()) to the specified partitions free memory list.

The function returns OK if it succeeds, else ERROR.

---

dm_free_uncached( )

Synopsis

```
void dm_free_uncached(
    void   *addr)     /* In: Pointer to the cache-safe buffer to deallocate. */
```

Description

Free the buffer acquired with dm_malloc_uncached()

This routine frees the buffer allocated by dm_malloc_uncached().

---

dm_malloc( )

Synopsis

```
INT16 dm_malloc(
    UINT8   identity,    /* In:  Identifier of the memory partition to allocate from. */
    UINT32  n_bytes,     /* In:  Number of bytes to allocate. */
    void    **pp_block)  /* Out: Pointer to the allocated block or NULL on failure. */
```

Description

Allocates memory from a specified partition. The size of the block will be equal to or greater than n_bytes.

If identity 0 is used, the allocation will be from the system memory pool.

Returns OK on success or ERROR if the identity is invalid or the memory can not be allocated.

---

dm_malloc_uncached( )

Synopsis

```
void * dm_malloc_uncached(
    UINT32   n_bytes)     /* In: Number of bytes to allocate. */
```

Description

Allocate a cache-safe buffer

This routine returns a pointer, aligned on a cache line boundary, that points to a section of memory that will not experience any cache coherency problems.

Returns a pointer to the cache-safe buffer, or NULL.

This function is not supported in CCU-O2 and will return NULL if used.

2.6 DR - Data Recorder Service

Overview

The CSS Data Recorder service provides a Data Recorder for sampling of signals in the application.

These are the features of the CSS Data Recorder:

- The sampling can be done periodical, event driven, synchronized with an application cyclic task or on demand from application task.
- The Data Recorder can be configured to trig on different conditions.
- The Data Recorder configuration is stored in files on the target file system.
- The Data Recorder is initialized according to the configuration found in the configuration files.
- The Data Recorder can be controlled with a set of RTS Monitor commands.
- Persistent memory is used to cope with unexpected situations e.g. if the target is reset or the power is turned off.
- Recordings are stored in files in the target file system. They can be uploaded to a host on demand.

DR Configuration

The DR service is configured using XML files. All configuration files are stored in the DR service directory on the target. The name of the DR service directory is ”dr” within the application root directory.

The DR service only starts if the application root directory points to a local file system (i.e. a file system on the target).

There is one DR service configuration file ”dr.conf” that holds a list of data recorders that are to be started during system initialization.

Each data recorder is configured using an XML file which is also stored in the DR service directory. The configuration files are named as the DR - with the extension ”.drc”. A DR name cannot contain white-space.

For more information about the configuration of the Data Recorder refer to: [UG]

DR Monitor Commands

The DR service provides an extensive RTS monitor command list. All DR monitor commands are prefixed by ”DR”. Typing ”DR” alone will show a help text.

For more information about the different RTS Monitor commands for the Data Recorder refer to: [MON]

---

dr_get_error_text( )

Synopsis

```
const char * dr_get_error_text(
    UINT32 dr_error)                                /* In: The error id */
```

Description

Return a pointer to the string that corresponds to the error code.
Returns NULL if the error code cannot be found in the string table.

---

dr_rec_config_get( )

Synopsis

```
INT16 dr_rec_config_get(
    const char * const dr_name,          /* In:  DR name */
    DR_REC_CFG         *p_conf )         /* Out: A copy of the configuration */
```

Description

Get a copy of the internal DR configuration.
The caller must supply memory for the copy.

Returns OK if the DR configuration is copied.
Returns ERROR if:
- The DR does not exist
- The pointer to the user supplied buffer is NULL

dr_rec_config_get2( )

Synopsis

```
INT16 dr_rec_config_get2(
    const char * const dr_name,  /* In:  DR name */
    DR_REC_CFG2        *p_conf ) /* Out: A copy of the configuration */
```

Description

Get a copy of the internal DR configuration.
The caller must supply memory for the copy.

This function is a complement to dr_rec_config_get when using functionality requiring extended structure DR_REC_CFG2.

Returns OK if the DR configuration is copied.
Returns ERROR if:
- The DR does not exist
- The pointer to the user supplied buffer is NULL

dr_rec_delete( )

Synopsis

```
INT16 dr_rec_delete(
    const char * const dr_name )                  /* In: DR name */
```

Description

Delete a DR.
This function does not delete any recording files. Use dr_rec_path_clean for this.

Returns OK if the DR was deleted.
Returns ERROR if:
- The DR does not exist
- DR is not stopped
- DR memory could not be freed

dr_rec_delete_file( )

Synopsis

```
INT16 dr_rec_delete_file(
    const char *filename)                     /* In: Name of recording file */
```

Description

Remove all files belonging to the designated recording file.
The DR services may internally use more than one file to store a recording. This function will remove all files belonging to the same recording, based on the argument filename.
Use API function dr_rec_find_file to find the file name.

Returns OK if all files were removed
Returns ERROR if:
- Error in file name
- File not found

dr_rec_exists( )

Synopsis

```
INT16 dr_rec_exists(
    const char * const dr_name )                    /* In: DR name */
```

Description

Return OK if the DR exists.
Returns ERROR if the DR does not exist.

dr_rec_find_file( )

Synopsis

```
INT16 dr_rec_find_file(
    const char *dr_name,                /* In:  DR name */
    BOOL        recover_flag,           /* In:  TRUE to find recovered files */
    unsigned    rec_no,                 /* In:  Recording number */
    unsigned    filename_len,           /* In:  Length of p_filename buffer */
    char       *p_filename)             /* Out: Name of recording file */
```

Description

This function searches the recording directory of the specified DR for a file with a name matching the given recover_flag and rec_no.
This first filename matching the recorder number is returned. Be aware that there MAY exist several recovered files with the same recording number.

Returns OK if the file could be found.
Returns ERROR if:
- DR does not exist
- File could not be found
- Buffer filename is too small to hold path+filename

dr_rec_find_file_first( )

Synopsis

```
INT16 dr_rec_find_file_first(
    const char *dr_name,                /* In:  DR name */
    BOOL        recover_flag,           /* In:  TRUE to find recovered files */
    unsigned    filename_len,           /* In:  Length of p_filename buffer */
    char       *p_filename)             /* Out: Name of recording file */
```

Description

This function returns the file name for the oldest Data Recording associated with the Data Recorder name.

The complete file name including the path and file extension is copied to the buffer p_filename provided by the caller.

Returns ERROR if:
- DR does not exist
- The parameters are invalid
- The file can not be found
- Buffer filename is too small to hold path+filename

dr_rec_find_file_next( )

Synopsis

```
INT16 dr_rec_find_file_next(
```

```
    const char *dr_name,              /* In:  DR name */
    BOOL        recover_flag,         /* In:  TRUE to find recovered files */
    const char *p_filename_in,        /* In:  Current file name */
    unsigned    filename_len,         /* In:  Length of p_filename buffer */
    char       *p_filename)           /* Out: Name of next recording file name */
```

Description

This function returns the file name for the next Data Recording associated with the Data Recorder name. The next recording is the recording following after the one specified in p_filename_in.

If the Data Recoder is running when this function is called the file name where the recording is currently being done is not returned.

The complete file name including the path and file extension is copied to the buffer p_filename provided by the caller.

Returns ERROR if:
- DR does not exist
- The parameters are invalid
- The input file name can not be accosiated with the DR
- The recover flag is TRUE and the input file name is not a recovered file
- The file can not be found
- Buffer filename is too small to hold path+filename

dr_rec_first( )

Synopsis

```
INT16 dr_rec_first(
    UINT32       dr_name_size,              /* In:  Size of name buffer */
    char * const dr_name_out)               /* Out: Name of first configured DR */
```

Description

Get the name of the first configured DR.

Returns OK if a DR was found.
Returns ERROR if:
- No DR was found
- The pointer to the out buffer is NULL
- The size of the out buffer is too small to hold the DR name

dr_rec_get_error( )

Synopsis

```
INT16 dr_rec_get_error(
    const char  *dr_name,                   /* In:  DR name */
    UINT32      *p_dr_error)                /* Out: The error code */
```

Description

Return the last found error info.

dr_rec_load( )

Synopsis

```
INT16 dr_rec_load(
    const char   * const dr_name,           /* In : DR name */
    DR_CFG_ERROR *       p_cfg_err )        /* Out: Error info */
```

Description

Create a recorder with the configuration from the DR configuration file.

Returns OK if the DR has been created.
Returns ERROR if:
- p_dr_cfg_err was NULL
- Could not open the configuration file for reading
- Error while parsing the configuration file
- DR name is empty
- The DR already exists
- Duplicate signal names
- Too small memory received
- Unable to allocate memory
- One or more configuration items is missing
- Two or more configuration items are inconsistent

If return code is ERROR then dr_cfg_err contains specific error information (unless p_dr_cfg_err was NULL).

---

dr_rec_next( )

Synopsis

```
INT16 dr_rec_next(
    const char * const dr_name,          /* In:  The DR name */
    UINT32       dr_name_size,           /* In:  Size of name buffer */
    char * const dr_name_out)            /* Out: Name of next configured DR */
```

Description

Get the name of the next configured DR.

Returns OK if a DR was found.
Returns ERROR if:
- The specified DR was not found
- End of the DR list
- The pointer to the out buffer is NULL
- The size of the out buffer is too small to hold the DR name

---

dr_rec_path_clean( )

Synopsis

```
INT16 dr_rec_path_clean(
    const char *dr_name)                         /* In: DR name */
```

Description

Clean the directory indicated by the path in the configuration.
This function only removes files that matches a DR file name pattern.
If the directory becomes empty then it is also removed.
The directory will not be removed if it contains other files than recording files or if it contains extra directories.

Returns OK if all the data recording files in the directory could be removed.
Returns ERROR if:
- Data recorder does not exist
- Data recorder is not stopped
- Data recording file could not be removed
- Data recording directory could not be removed - allthough it is empty.

---

dr_rec_start( )

Synopsis

```
INT16 dr_rec_start(
    const char * const dr_name )                 /* In: DR name */
```

Description

Start a DR.

Returns OK if the DR has been started.
Returns ERROR if:
- The DR does not exist
- The DR is not stopped
- One or more configuration items is missing
- Two or more configuration items are inconsistent
- A periodic task could not be created (CLOCK PERIOD)
- The DR could not be attached to the periodic task.

---

dr_rec_started( )

Synopsis

```
INT16 dr_rec_started(
    const char * const dr_name )                    /* In: DR name */
```

Description

Get the state of a DR.

Return OK if the DR is running.
Returns ERROR if:
- The DR does not exist
- The DR is not started.

---

dr_rec_stop( )

Synopsis

```
INT16 dr_rec_stop(
    const char * const dr_name )                    /* In: DR name */
```

Description

Stop a DR.

Returns OK if the DR has been stopped.
Returns ERROR if:
- The DR does not exist
- The DR is not started
- The sample function could not be unhooked
- The sample task could not terminate.

---

dr_rec_trig( )

Synopsis

```
INT16 dr_rec_trig(
    const char * const dr_name )                    /* In: DR name */
```

Description

Trigger a DR.
If the DR is an event recorder then one sample is taken by the DR.
If the DR is a transient recorder then a complete recording is made.

Returns OK if the recorder was triggered
Returns FALSE if:
- DR is not a transient or event recorder

- DR does not exists
- The DR is neither an event nor a transient recorder

dr_rec_upload_file( )

Synopsis

```
INT16 dr_rec_upload_file(
    const char  *file_name,     /* In: Recording file name */
    const char  *hostname,      /* In: FTP Server host name or IP address */
    const char  *user,          /* In: FTP server user name */
    const char  *passwd,        /* In: FTP server password */
    const char  *acct,          /* In: FTP server account (usually "") */
    const char  *dirname)       /* In: Relative path to host directory */
```

Description

Transfer a complete recording to an FTP server.
Use API function dr_rec_find_file to find the file name.

Returns OK if the file was successfully transfered
Returns ERROR if:
- Illegal file name
- Recording could not be found
- An FTP connection could not be established
- FTP server could not be found
- Bad user name, password or account information
- Error writing to the FTP connection
- Error closing the FTP connection

dr_signal_config_get( )

Synopsis

```
INT16 dr_signal_config_get(
    const char * const dr_name,     /* In:  DR name */
    const char * const sig_name,    /* In:  Signal name */
    DR_SIGNAL_CFG     *p_sig_cfg ) /* Out: A copy of the signal configuration */
```

Description

Get a copy of the internal signal configuration.
The caller must supply memory for the copy.

Returns OK if the signal configuration is copied.
Returns ERROR if:
- The DR does not exist
- The signal is not attached to the DR
- The pointer to the user supplied buffer is NULL

dr_signal_config_get2( )

Synopsis

```
INT16 dr_signal_config_get2(
    const char * const dr_name,     /* In:  DR name */
    const char * const sig_name,    /* In:  Signal name */
    DR_SIGNAL_CFG2    *p_sig_cfg ) /* Out: A copy of the signal configuration */
```

Description

Get a copy of the internal signal configuration.
The caller must supply memory for the copy.

This function is a complement to dr_signal_config_get when using functionality requiring extended structure DR_SIGNAL_CFG2.

Returns OK if the signal configuration is copied.
Returns ERROR if:
- The DR does not exist
- The signal is not attached to the DR
- The pointer to the user supplied buffer is NULL

---

dr_signal_first( )

Synopsis

```
INT16 dr_signal_first(
    const char * const dr_name,          /* In:  DR name */
    UINT32       sig_name_size,          /* In:  sizeof(sig_name) */
    char * const sig_name )              /* Out: A copy of the signal name */
```

Description

Get a copy of the first attached signal of a DR.

Returns OK if the signal name has been returned.
Returns ERROR if:
- DR does not exist
- DR does not have any signals attached
- Length of first signal name is larger than sig_name_size

---

dr_signal_next( )

Synopsis

```
INT16 dr_signal_next(
    const char * const dr_name,              /* In:  DR name */
    const char * const sig_name_in,          /* In:  Current signal name */
    UINT32       sig_name_size,              /* In:  sizeof(sig_name) */
    char * const sig_name_out )              /* OUT: Name of next signal */
```

Description

Get a copy of the name of the next signal in the signal list.

Returns OK if the signal name was returned.
Returns ERROR if:
- The DR does not exist
- The given signal is not attached to the DR
- Length of next signal name is larger than sig_name_size

2.7 DRRCS - Data Recorder Reconfiguration Service

Overview

The Data Recorder Reconfiguration Service provides support to change the configuration of a specific Data Recorder during RUN mode. The service communicates with the MCG according to the File Download Data protocol. For information about the communication with the MCG see: [MCGEDSICD]

These are the features of the Data Recorder Reconfiguration Service:

- Support for downloading of a new DR configuration via MCG file transfer protocol
- Verification of the new configuration before reloading regarding XML syntax and CRC
- Checking that the new DR configuration is not violating rules and limitations
- Installing the new configuration in DVS
- Reloading the new Configuration in the DR service
- Reporting the download and installation result back to the MCG

DR Configuration limitations and rules

A downloaded DR configuration replacing a present configuration must follow certain rules and limitations to be accepted for reloading. Some DR XML configuration parameters are not allowed the be changed. The rules are specified in the present loaded configuration and are mandatory. The mandatory parameters are not allowed to be changed. The following DR XML configuration parameters defines the rules:

- DR_REMOTE_MOD_ALLOWED - specifies if the DR is allowed to be reloaded - mandatory
- DR_MAX_SIGNALS - specifies how many signals are allowed in the DR configuration - mandatory
- DR_MIN_CLOCK_CYCLE - specifies the fastest task cycle allowed - manatory if DR_CLOCK_MODE is CYCLIC
- DR_CLOCK_CYCLIC_PRIORITY - is not allowed to be modified if DR_CLOCK_MODE is CYCLIC
- DR_CLOCK_TASK_NAME - is not allowed to be modified if DR_CLOCK_MODE is TASK
- DR_USE_RAM_DISK - is not allowed to be modified if RAM disk is used
- DR_MAX_SIGNALS - is not allowed to be modified
- DR_NAME - is not allowed to be modified
- DR_FILES_PATH - is not allowed to be modified

For more information about the configuration of the Data Recorder refer to: [UG]

The Data Recorder Reconfiguration Service is initialized in RUN mode if the DR is running. By default the service is disabled and will only respond that it is in disabled state to any request. To enable the service the API function drrcs_reconf_service_enable( ) can be called from the application.

---

drrcs_reconf_service_enable( )

Synopsis

```
void drrcs_reconf_service_enable(
    BOOL enable)        /* In: FALSE to disable or TRUE to enable */
```

Description

This function enables or disables the Data Recorder Reconfiguration Service.
The function can be used by the application to control the access to the reconfiguration service.
If a send file request is recieved when the service is disabled the application result code will be DL_SRV_TEMP_UNAVAILABLE (2) in the response and status message.
Default state for the Data Recorder Reconfiguration Service is disabled.
This function is only available in the RTS-CSSEXT variant of CSS.

2.8 DVS - Download and Versioning System Service

This service integrates the DVS (Download and Versioning System) library in CSS.

This section includes a description of the DVS API.

The following definitions and typedefs are used throughout the DVS API:

```
#define DVS_IPC_DEFAULT_PORT_NO             40127
#define DVS_IPC_CALLBACK_DEFAULT_PORT_NO    (DVS_IPC_DEFAULT_PORT_NO + 1)

#define DVS_IPC_INVALID_ULU_ADDRESS         0xFFFFFFFF

#define DVS_IFC_OK                          0
#define DVS_IFC_ERROR                       1
#define DVS_IFC_BUFFER_OVFL                 2
#define DVS_IFC_OUT_OF_MEMORY               3
#define DVS_IFC_NOT_OPENED                  4
#define DVS_IFC_ALREADY_OPENED              5
#define DVS_IFC_NO_SERVER                   6
#define DVS_IFC_NO_SOCKET                   7
#define DVS_IFC_SOCKET_ERROR                8
#define DVS_IFC_TIMEOUT                     9
#define DVS_IFC_SOCKET_CLOSED               10
#define DVS_IFC_CONNECT_FAILED              11
#define DVS_IFC_DLU_NODATA                  12
#define DVS_IFC_DLU_BAD_CRC                 13
#define DVS_IFC_INV_TRANSITION              14
#define DVS_IFC_NO_DLU                      15
#define DVS_IFC_CBK_ALREADY_ASSIGNED        16
#define DVS_IFC_DONT_CLEANUP                17
#define DVS_IFC_NO_ULU                      18
#define DVS_IFC_NO_HWINFO                   19
#define DVS_IFC_NO_FILE_SYSTEM              20
#define DVS_IFC_NO_SWINFO                   21
#define DVS_IFC_ALREADY_LOCKED              22
#define DVS_IFC_INV_CLIENT_HANDLE           23
#define DVS_IFC_FCREATE_FAILED              24
#define DVS_IFC_DISK_FULL                   25
#define DVS_IFC_INV_FILE_HANDLE             26
#define DVS_IFC_WR_SEQ_ERROR                27
#define DVS_IFC_CRC_NOT_VALIDATED           28
#define DVS_IFC_NO_MORE_FILES               29

#define DVS_IFC_STATE_IDLE                  0
#define DVS_IFC_STATE_ACTIVATED             1
#define DVS_IFC_STATE_RUNNING               2
#define DVS_IFC_STATE_LASTTIMERUN           3
#define DVS_IFC_STATE_UNKNOWN               4

#define DVS_IFC_STATE_MASK_ALL              0xFFFFFFFF
#define DVS_IFC_STATE_MASK_IDLE             0x00000001
#define DVS_IFC_STATE_MASK_ACTIVATED        0x00000002
#define DVS_IFC_STATE_MASK_RUNNING          0x00000004
#define DVS_IFC_STATE_MASK_LASTTIMERUN      0x00000008



#define DVS_IFC_OPM_UNKNOWN                 0x00000000
#define DVS_IFC_OPM_BOOTLOADER              0x00000001
#define DVS_IFC_OPM_OS_IDLE                 0x00000002
```

```
#define DVS_IFC_OPM_FULL                    0x00000003
#define DVS_IFC_OPM_DL_INHIBIT              0x00000004
#define DVS_IFC_OPM_BMS                     0x00000005
#define DVS_IFC_OPM_FULL_NODL               0x00000006
#define DVS_IFC_OPM_APPL_LESS               0x00000007
#define DVS_IFC_OPM_APPL_SUSP               0x00000008



#define DVS_IFC_CLEANUP_MODE_APPLWARE       0x00000000
#define DVS_IFC_CLEANUP_MODE_FULL           0x00000001
#define DVS_IFC_CLEANUP_MODE_GARBAGE        0x00000002

#define DVS_IFC_FLAG_MULTI_VERSION_SUPPORT  0x00000001

#define DVS_IFC_VIRT_ADDR_UNKNOWN           0xFFFFFFFF

#define DLU_NAME_SIZE                       32
#define DLU_PACKAGE_IDENT_SIZE              8

#define ULU_TARGET_HW_TYPE_SIZE             32
#define ULU_HEADER_RESERVE                  24
#define ULU_HEADER_LENGTH                   60

#define DLU_TARGET_PATH_SIZE                1024
#define DLU_TPATH_HEADER_LENGTH             (DLU_TARGET_PATH_SIZE + 12)

typedef int DVS_RESULT;

typedef DVS_RESULT (TYPE_DVS_CLEANUP_IND_PROC) (void);

typedef struct STR_DLU_PROPERTY
{
    char    dluName[33];
    UINT32  dluVersion;
    UINT32  dluState;
    UINT32  dluLastState;
    UINT32  dluTypeCode;
    UINT32  fileRefSize;
    char    *fileRef;
} DLU_PROPERTY;

typedef struct STR_DLU_PROPERTY_EXT
{
    char    dluName[33];
    UINT32  dluVersion;
    UINT32  dluState;
    UINT32  dluLastState;
    UINT32  dluTypeCode;
    UINT32  dluCrc;
    UINT32  dluSize;
    UINT32  fileRefSize;
    UINT32  reserved[0x20];
    char    *fileRef;
} DLU_PROPERTY_EXT;

typedef struct STR_DLU_PROPERTY_EXT2
{
    char    dluName[33];
    UINT32  dluVersion;
```

```
    UINT32  dluState;
    UINT32  dluLastState;
    UINT32  dluTypeCode;
    UINT32  dluCrc;
    UINT32  dluSize;
    UINT32  dluTimeStamp;
    UINT32  dluVirtualAddress;
    UINT32  dluPermVal;
    UINT32  fileRefSize;
    UINT32  hostFileNameSize;
    UINT32  reserved[0x20];
    char    *fileRef;
    char    *hostFileName;
} DLU_PROPERTY_EXT2;

typedef struct STR_DLU_PROPERTY_EXT3
{
    char    dluName[33];
    UINT32  dluVersion;
    UINT32  dluState;
    UINT32  dluLastState;
    UINT32  dluTypeCode;
    UINT32  dluCrc;
    UINT32  dluSize;
    UINT32  dluTimeStamp;
    UINT32  dluVirtualAddress;
    UINT32  dluPermVal;
    UINT32  fileRefSize;
    UINT32  hostFileNameSize;
    char    edPackName[32];
    UINT32  edPackVersion;
    char    sciName[32];
    UINT32  sciVersion;
    char    packageSource[3];
    char    fillByte;
    UINT32  reserved[0x20];
    char    *fileRef;
    char    *hostFileName;
} DLU_PROPERTY_EXT3;

typedef struct STR_DLU_PROPERTY_EXT4
{
    char    dluName[33];
    UINT32  dluVersion;
    UINT32  dluState;
    UINT32  dluLastState;
    UINT32  dluTypeCode;
    UINT32  dluCrc;
    UINT32  dluSize;
    UINT32  dluTimeStamp;
    UINT32  dluVirtualAddress;
    UINT32  dluPermVal;
    UINT32  fileRefSize;
    UINT32  hostFileNameSize;
    char    edPackName[32];
    UINT32  edPackVersion;
    char    sciName[32];
    UINT32  sciVersion;
    char    packageSource[3];
    char    fillByte;
```

```
    UINT32  reserved[0x20];
    char    *fileRef;
    char    *hostFileName;
    UINT32  uluBaseAddress;
    UINT32  edCrc;
    char    edInfo[32];
    char    edTimeStamp[32];
    UINT32  sciCrc;
    char    sciInfo[32];
    char    sciTimeStamp[32];
    UINT8   edHasCrc;
    UINT8   sciHasCrc;
} DLU_PROPERTY_EXT4;

typedef struct STR_DLU_PROPERTY_EXT5
{
    char    dluName[33];
    UINT32  dluVersion;
    UINT32  dluState;
    UINT32  dluLastState;
    UINT32  dluTypeCode;
    UINT32  dluCrc;
    UINT32  dluSize;
    UINT32  dluTimeStamp;
    UINT32  dluVirtualAddress;
    UINT32  dluPermVal;
    UINT32  fileRefSize;
    UINT32  hostFileNameSize;
    char    edPackName[128];
    UINT32  edPackVersion;
    char    sciName[128];
    UINT32  sciVersion;
    char    packageSource[3];
    char    fillByte;
    UINT32  reserved[0x20];
    char    *fileRef;
    char    *hostFileName;
    UINT32  uluBaseAddress;
    UINT32  edCrc;
    char    edInfo[128];
    char    edTimeStamp[32];
    UINT32  sciCrc;
    char    sciInfo[128];
    char    sciTimeStamp[32];
    UINT8   edHasCrc;
    UINT8   sciHasCrc;
} DLU_PROPERTY_EXT5;

typedef struct STR_ULU_PROPERTY
{
    char    uluName[33];
    UINT32  uluAddress;
    UINT32  uluSize;
    UINT32  uluVersion;
} ULU_PROPERTY;

typedef struct STR_ULU_PROPERTY_EXT
{
    char    uluName[33];
    UINT32  uluAddress;
```

```
    UINT32  uluSize;
    UINT32  uluCrc;
    UINT32  uluVersion;
    UINT32  reserved[0x20];
} ULU_PROPERTY_EXT;

typedef struct STR_DLU_HEADER
{
    CHAR8   packageIdent[DLU_PACKAGE_IDENT_SIZE];
    UINT32  headerLength;
    UINT32  headerCrc32;
    UINT32  headerVersion;
    UINT32  dluVersion;
    UINT32  dluDataSize;
    UINT32  dluTypeCode;
    UINT32  dluCrc32;
    CHAR8   dluName[DLU_NAME_SIZE];
} TYPE_DLU_HEADER;

typedef struct STR_DLU_ULU_HEADER
{
    UINT32  targetAddress;
    CHAR8   targetHWType[ULU_TARGET_HW_TYPE_SIZE];
    CHAR8   reserve[ULU_HEADER_RESERVE];
} TYPE_DLU_ULU_HEADER;

typedef struct STR_DLU_TPATH_HEADER
{
    UINT32  usePermValue;
    UINT32  useTargetPath;
    UINT32  permValue;
    CHAR8   targetPath[DLU_TARGET_PATH_SIZE];
} TYPE_DLU_TPATH_HEADER;
```

dvs_assign_cleanup_callback( )

Synopsis

```
DVS_RESULT dvs_assign_cleanup_callback(
    TYPE_DVS_CLEANUP_IND_PROC cleanupIndication)
```

Description

Purpose:

The dvs_assign_cleanup_callback() function allows to assign an application specific callback handler which is performed, whenever a MCP/MCPM cleanup request is detected by DVS. By help of this callback, the client software which makes use of the DVS interface may perform all necessary activities which are required, before DVS actually removes all applications from the target's persistent storage.

Parameters:

cleanupIndication	The cleanupIndication parameter holds a pointer to a callback routine provided by the caller of this function. If cleanupIndication is a NULL pointer, a formerly performed callback assignment will be removed. The cleanup indication is provided for one and only one DVS interface client at the same time, i.e. if the assignment is performed more than once, no new assignment will be done and a corresponding error code will indicate this situation.

Return values:

DVS_IFC_OK:	The callback could be assigned successfully. Whenever a cleanup request is detected, the user supplied callback routine will be performed (if cleanupIndication is not NULL).
DVS_IFC_CBK_ALREADY_ASSIGNED:	There is already a callback assigned, so the new assignment has not been done.

Remark:

The user supplied callback handler shall return one of the following result codes:

DVS_IFC_OK:	The callback handler has performed successfully all mandatory activities which have to be done before cleanup. DVS may perform the cleanup after return from the cleanup indication handler.
DVS_IFC_DONT_CLEANUP:	The DVS interface client explicitly wants to suppress the DVS cleanup operation.

Other return codes shall not be used, but if any other code is returned, DVS will handle them as DVS_IFC_OK, i.e. the cleanup operation will actually be performed.

---

dvs_close( )

Synopsis

```
DVS_RESULT dvs_close(
    void)
```

Description

Purpose:

A call of the dvs_open() routine might allocate some operating system and communication resources. Any application which successfully performed the dvs_open() shall explicitly call the dvs_close() routine in order to release all those allocated resources.

Parameters:

None

Return values:

DVS_IFC_OK:	The DVS interface has been closed successfully. All resources allocated by the former dvs_open() are released.

Remark:

None

---

dvs_delete_dlu( )

Synopsis

```
DVS_RESULT dvs_delete_dlu(
    char     *dluname,
    UINT32   dluversion)
```

Description

Purpose:

The function dvs_delete_dlu() allows the removal of a DLU from the persistent storage of the target device. The DLU to be removed must be specified explicitly by its unique identification which consists of the DLU name and the DLU version.

Parameters:

dluname:	The name of the DLU to be deleted. The dluname parameter is case sensitive, i.e. the dluname must be specified exactly in the way which is used within the internal data structures of the DLU itself.
dluversion:	The version of the DLU to be deleted.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The DLU specified by dluname and dluversion could be removed successfully from targets persistent storage.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_NO_DLU:	The parameter dluname together with the parameter dluversion do not reference a DLU currently existing on the device.

Remark:

None

dvs_disable_reboot( )

Synopsis

```
DVS_RESULT dvs_disable_reboot(
    void)
```

Description

Purpose:

The dvs_disable_reboot() and dvs_enable_reboot() functions provide means for disabling and enabling the forced reboot functionality of the DVS during runtime.

Parameters: None

Return values:

DVS_IFC_OK:	The request could be performed successfully. The forced reboot functionality of the DVS is blocked until the lock is explicitly removed by calling the dvs_enable_reboot() function.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.

Remark:

None

dvs_enable_reboot( )

Synopsis

```
DVS_RESULT dvs_enable_reboot(
    void)
```

Description

Purpose:

The dvs_disable_reboot() and dvs_enable_reboot() functions provide means for disabling and enabling the forced reboot functionality of the DVS during runtime.

Parameters: None

Return values:

DVS_IFC_OK:	The request could be performed successfully. The forced reboot functionality of the DVS is re-enabled again.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.

Remark:

None

---

dvs_get_dlu_fileref( )

Synopsis

```
DVS_RESULT dvs_get_dlu_fileref(
    char            *dluname,
    UINT32          dluVersionVal,
    UINT32          fileRefIndex,
    char            *pFilePathBuffer,
    UINT32          *pBufferLen)
```

Description

Purpose:

A DLU is of type DLU_TYPE_LINUX_TAR contains an archive, which in turn may contain one or more files. The dvs_get_dlu_fileref() function may be used to get references to files within those TAR files.

The references may then be used to retrieve DLU information about files within the TAR archive, using the dvs_get_dlu_info() function.

Parameters:

dluname	The name of the DLU file.
dluVersionVal	The parameters dluVersionVal allow using the DLU version as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true: DLUs version == dluVersionVal
fileRefIndex	This number specifies which file within the TAR file to address.
pFilePathBuffer	Holds a pointer to a caller-supplied buffer where the name of the referenced file is stored by dvs_get_dlu_fileref.
pBufferLen	If the DLU info buffer is provided by the caller of the dvs_get_dlu_fileref() routine, the pBufferLen parameter has to specify the size of this user provided buffer. If the DLU info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The buffer pointed to by pFilePathBuffer holds the name of the referenced file within the TAR archieve.
DVS_IFC_ERROR:	Invalid parameter specified (pFilePathBuffer or pBufferLen is a NULL pointer).
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_BUFFER_OVFL:	The static data buffer provided by the caller of the routine is not sufficient to hold the complete file reference.
DVS_IFC_NO_MORE_FILES:	The DLU does not exist in the DVS repository or is invalid or the end of the list of files in the TAR file has been reached.

Remark:

None.

---

dvs_get_dlu_info( )

Synopsis

```
DVS_RESULT dvs_get_dlu_info(
    char           *dluname,
    UINT32         dluVersionVal,
    UINT32         dluVersionMask,
    UINT32         stateflags,
    UINT32         typecode,
    UINT32         typemask,
    UINT32         bufferSize,
    UINT32         *pCount,
    DLU_PROPERTY   **ppBuffer)
```

Description

Purpose:

The dvs_get_dlu_info() function provides access to version information of all DLUs currently installed on target device.

The function may be used with static buffer provided by the caller of the routine as well as with dynamic buffer provided by the interface itself. In the latter case, the buffer provided by the dvs_get_dlu_info() routine must be freed explicitly by a call of the dvs_release_dlu_info_buffer().

Parameters:

dluname: The parameter dluname allows to restrict the search strategy of the dvs_get_dlu_info() routine to DLUs with the explicitly specified DLU name.
If the dluname parameter is an empty string or the dluname parameter is a NULL pointer, the name of the DLU is no filter criterion, i.e. dvs_get_dlu_info() will deliver the information of DLUs independent from their name.

dluVersionVal:
dluVersionMask: The parameters dluVersionVal as well as dluVersionMask allow using the DLU version as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true:

DLUs version AND dluVersionMask == dluVersionVal AND dluVersionMask

i.e. a dluVersionMask of zero will deliver all DLUs independent from their actual version.

stateflags: The stateflags parameter allows a restrictive query on DLUs within a specific state. The DVS_IFC_STATE_MASK_xxxx symbolic constants shall be used to define the expected DLU state filter criterion. The filter value is defined as bit field, i.e. an 'ored' combination of several state constants may be used.
The specific constant DVS_IFC_STATE_MASK_ALL shall be used, all DLUs independent from the actual current state shall be delivered.

typecode:
typemask: The parameters typecode as well as typemask allow to use the DLU type code as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true:

DLUs type code AND typemask == typecode AND typemask

i.e. a typemask of zero will deliver all DLUs independent from their actual type code.

bufferSize: If the DLU info buffer is provided by the caller of the dvs_dlu_get_info() routine, the bufferSize parameter has to specify the size of this user provided buffer. If the DLU info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.

pCount: The parameter pCount holds the pointer to a user provided variable which will hold the actual number of retrieved DLUs after successful execution of the function.

ppBuffer: The parameter ppBuffer holds the pointer to a user provided variable which holds the pointer to the retrieved DLU info data buffer.
If the dvs_dlu_get_info() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppBuffer shall be initialized with the pointer to that buffer before the routine is called.
If the memory for the DLU info buffer shall be allocated dynamically by the interface itself, the ppBuffer variable shall be initialized by a NULL pointer before the routine is called. In the latter case, the ppBuffer variable will hold the pointer to the dynamically allocated buffer on successful return from dvs_dlu_get_info().

Return values:

DVS_IFC_OK: The request could be performed successfully.
The variable pointed to by ppBuffer holds the pointer to a data block, which holds all the retrieved DLU version information.

DVS_IFC_ERROR: Invalid parameter specified (pCount is a NULL pointer).

DVS_IFC_NOT_OPENED: The function could not be performed as the interface has not been opened before.

DVS_IFC_OUT_OF_MEMORY: The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.

DVS_IFC_BUFFER_OVFL: The static data buffer provided by the caller of the routine is not sufficient to hold the complete DLU info data structure.

DVS_IFC_NO_DLU: There are currently no DLUs available or the filter criteria are defined in a way so that no DLU passes the filter. In case of dynamic memory management, no data buffer is actually allocated and thus no memory buffer needs to be released.

Remark:

It should be kept in mind, that the required buffer size for given maximum number of DLU items is not the same as sizeof(DLU_PROPERTY) * maximum number of DLUs. The reason for this is that the actual string data is not part of the DLU_PROPERTY structure. There is just a pointer to the string in. The actual string data needs to be copied behind the DLU info array, so that the required buffer size is typically much bigger than the pure array size. To get rid of the need for buffer size estimation on application side, the usage of dynamic buffer handling is recommended.

If dynamic buffer management is chosen (i.e. *ppBuffer is NULL and bufferSize is 0), the caller of the routine is responsible that the DLU data buffer is released by use of the dvs_release_dlu_info_buffer() function.

---

dvs_get_dlu_info_ext( )

Synopsis

```
DVS_RESULT dvs_get_dlu_info_ext(
    char               *dluname,
    UINT32             dluVersionVal,
    UINT32             dluVersionMask,
    UINT32             stateflags,
    UINT32             typecode,
    UINT32             typemask,
    UINT32             bufferSize,
    UINT32             *pCount,
    DLU_PROPERTY_EXT   **ppBuffer)
```

Description

Purpose:

The dvs_get_dlu_info_ext() function provides access to version information of all DLUs currently installed on target device.

The function may be used with static buffer provided by the caller of the routine as well as with dynamic buffer provided by the interface itself. In the latter case, the buffer provided by the dvs_get_dlu_info_ext() routine must be freed explicitly by a call of the dvs_release_dlu_info_buffer_ext().

Parameters:

dluname: The parameter dluname allows to restrict the search strategy of the dvs_get_dlu_info_ext() routine to DLUs with the explicitly specified DLU name.
If the dluname parameter is an empty string or the dluname parameter is a NULL pointer, the name of the DLU is no filter criterion, i.e. dvs_get_dlu_info_ext() will deliver the information of DLUs independent from their name.

dluVersionVal:
dluVersionMask: The parameters dluVersionVal as well as dluVersionMask allow using the DLU version as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true:

DLUs version AND dluVersionMask == dluVersionVal AND dluVersionMask

i.e. a dluVersionMask of zero will deliver all DLUs independent from their actual version.

stateflags: The stateflags parameter allows a restrictive query on DLUs within a specific state. The DVS_IFC_STATE_MASK_xxxx symbolic constants shall be used to define the expected DLU state filter criterion. The filter value is defined as bit field, i.e. an 'ored' combination of several state constants may be used.
The specific constant DVS_IFC_STATE_MASK_ALL shall be used, all DLUs independent from the actual current state shall be delivered.

typecode:
typemask: The parameters typecode as well as typemask allow to use the DLU type code as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true:

DLUs type code AND typemask == typecode AND typemask

i.e. a typemask of zero will deliver all DLUs independent from their actual type code.

bufferSize: If the DLU info buffer is provided by the caller of the dvs_dlu_get_info_ext() routine, the bufferSize parameter has to specify the size of this user provided buffer. If the DLU info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.

pCount: The parameter pCount holds the pointer to a user provided variable which will hold the actual number of retrieved DLUs after successful execution of the function.

ppBuffer: The parameter ppBuffer holds the pointer to a user provided variable which holds the pointer to the retrieved DLU info data buffer.
If the dvs_dlu_get_info_ext() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppBuffer shall be initialized with the pointer to that buffer before the routine is called.
If the memory for the DLU info buffer shall be allocated dynamically by the interface itself, the ppBuffer variable shall be initialized by a NULL pointer before the routine is called. In the latter case, the ppBuffer variable will hold the pointer to the dynamically allocated buffer on successful return from dvs_dlu_get_info_ext().

Return values:

DVS_IFC_OK: The request could be performed successfully.
The variable pointed to by ppBuffer holds the pointer to a data block, which holds all the retrieved DLU version information.

DVS_IFC_ERROR: Invalid parameter specified (pCount is a NULL pointer).

DVS_IFC_NOT_OPENED: The function could not be performed as the interface has not been opened before.

DVS_IFC_OUT_OF_MEMORY: The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.

DVS_IFC_BUFFER_OVFL: The static data buffer provided by the caller of the routine is not sufficient to hold the complete DLU info data structure.

DVS_IFC_NO_DLU: There are currently no DLUs available or the filter criteria are defined in a way so that no DLU passes the filter. In case of dynamic memory management, no data buffer is actually allocated and thus no memory buffer needs to be released.

Remark:

It should be kept in mind, that the required buffer size for given maximum number of DLU items is not the same as sizeof(DLU_PROPERTY_EXT) * maximum number of DLUs. The reason for this is that the actual string data is not part of the DLU_PROPERTY_EXT structure. There is just a pointer to the string in. The actual string data needs to be copied behind the DLU info array, so that the required buffer size is typically much bigger than the pure array size. To get rid of the need for buffer size estimation on application side, the usage of dynamic buffer handling is recommended.

If dynamic buffer management is chosen (i.e. *ppBuffer is NULL and bufferSize is 0), the caller of the routine is responsible that the DLU data buffer is released by use of the dvs_release_dlu_info_buffer_ext() function.

---

dvs_get_dlu_info_ext2( )

Synopsis

```
DVS_RESULT dvs_get_dlu_info_ext2(
    char                *dluname,
    UINT32              dluVersionVal,
    UINT32              dluVersionMask,
    UINT32              stateflags,
    UINT32              typecode,
    UINT32              typemask,
    UINT32              bufferSize,
    UINT32              *pCount,
    DLU_PROPERTY_EXT2   **ppBuffer)
```

Description

Purpose:

The dvs_get_dlu_info_ext2() function provides access to version information of all DLUs currently installed on target device.

The function may be used with static buffer provided by the caller of the routine as well as with dynamic buffer provided by the interface itself. In the latter case, the buffer provided by the dvs_get_dlu_info_ext2() routine must be freed explicitly by a call of the dvs_release_dlu_info_buffer_ext2().

Parameters:

dluname: The parameter dluname allows to restrict the search strategy of the dvs_get_dlu_info_ext2() routine to DLUs with the explicitly specified DLU name.
If the dluname parameter is an empty string or the dluname parameter is a NULL pointer, the name of the DLU is no filter criterion, i.e. dvs_get_dlu_info_ext2() will deliver the information of DLUs independent from their name.

dluVersionVal:
dluVersionMask: The parameters dluVersionVal as well as dluVersionMask allow using the DLU version as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true:

DLUs version AND dluVersionMask == dluVersionVal AND dluVersionMask

i.e. a dluVersionMask of zero will deliver all DLUs independent from their actual version.

stateflags: The stateflags parameter allows a restrictive query on DLUs within a specific state. The DVS_IFC_STATE_MASK_xxxx symbolic constants shall be used to define the expected DLU state filter criterion. The filter value is defined as bit field, i.e. an 'ored' combination of several state constants may be used.
The specific constant DVS_IFC_STATE_MASK_ALL shall be used, all DLUs independent from the actual current state shall be delivered.

typecode:
typemask: The parameters typecode as well as typemask allow to use the DLU type code as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true:

DLUs type code AND typemask == typecode AND typemask

i.e. a typemask of zero will deliver all DLUs independent from their actual type code.

bufferSize: If the DLU info buffer is provided by the caller of the dvs_dlu_get_info_ext2() routine, the bufferSize parameter has to specify the size of this user provided buffer. If the DLU info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.

pCount: The parameter pCount holds the pointer to a user provided variable which will hold the actual number of retrieved DLUs after successful execution of the function.

ppBuffer: The parameter ppBuffer holds the pointer to a user provided variable which holds the pointer to the retrieved DLU info data buffer.
If the dvs_dlu_get_info_ext2() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppBuffer shall be initialized with the pointer to that buffer before the routine is called.
If the memory for the DLU info buffer shall be allocated dynamically by the interface itself, the ppBuffer variable shall be initialized by a NULL pointer before the routine is called. In the latter case, the ppBuffer variable will hold the pointer to the dynamically allocated buffer on successful return from dvs_dlu_get_info_ext2().

Return values:

DVS_IFC_OK: The request could be performed successfully.
The variable pointed to by ppBuffer holds the pointer to a data block, which holds all the retrieved DLU version information.

DVS_IFC_ERROR: Invalid parameter specified (pCount is a NULL pointer).

DVS_IFC_NOT_OPENED: The function could not be performed as the interface has not been opened before.

DVS_IFC_OUT_OF_MEMORY: The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.

DVS_IFC_BUFFER_OVFL: The static data buffer provided by the caller of the routine is not sufficient to hold the complete DLU info data structure.

DVS_IFC_NO_DLU: There are currently no DLUs available or the filter criteria are defined in a way so that no DLU passes the filter. In case of dynamic memory management, no data buffer is actually allocated and thus no memory buffer needs to be released.

Remark:

It should be kept in mind, that the required buffer size for given maximum number of DLU items is not the same as sizeof(DLU_PROPERTY_EXT2) * maximum number of DLUs. The reason for this is that the actual string data is not part of the DLU_PROPERTY_EXT2 structure. There is just a pointer to the string in. The actual string data needs to be copied behind the DLU info array, so that the required buffer size is typically much bigger than the pure array size. To get rid of the need for buffer size estimation on application side, the usage of dynamic buffer handling is recommended.

If dynamic buffer management is chosen (i.e. *ppBuffer is NULL and bufferSize is 0), the caller of the routine is responsible that the DLU data buffer is released by use of the dvs_release_dlu_info_buffer_ext2() function.

---

dvs_get_dlu_info_ext3( )

Synopsis

```
DVS_RESULT dvs_get_dlu_info_ext3(
    char                *dluname,
    UINT32              dluVersionVal,
    UINT32              dluVersionMask,
    UINT32              stateflags,
    UINT32              typecode,
    UINT32              typemask,
    UINT32              bufferSize,
    UINT32              *pCount,
    DLU_PROPERTY_EXT3   **ppBuffer)
```

Description

Purpose:

The dvs_get_dlu_info_ext3() function provides access to version information of all DLUs currently installed on target device.

The function may be used with static buffer provided by the caller of the routine as well as with dynamic buffer provided by the interface itself. In the latter case, the buffer provided by the dvs_get_dlu_info_ext3() routine must be freed explicitly by a call of the dvs_release_dlu_info_buffer_ext3().

Parameters:

dluname: The parameter dluname allows to restrict the search strategy of the dvs_get_dlu_info_ext3() routine to DLUs with the explicitly specified DLU name.
If the dluname parameter is an empty string or the dluname parameter is a NULL pointer, the name of the DLU is no filter criterion, i.e. dvs_get_dlu_info_ext3() will deliver the information of DLUs independent from their name.

dluVersionVal:
dluVersionMask: The parameters dluVersionVal as well as dluVersionMask allow using the DLU version as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true:

DLUs version AND dluVersionMask == dluVersionVal AND dluVersionMask

i.e. a dluVersionMask of zero will deliver all DLUs independent from their actual version.

stateflags: The stateflags parameter allows a restrictive query on DLUs within a specific state. The DVS_IFC_STATE_MASK_xxxx symbolic constants shall be used to define the expected DLU state filter criterion. The filter value is defined as bit field, i.e. an 'ored' combination of several state constants may be used.
The specific constant DVS_IFC_STATE_MASK_ALL shall be used, all DLUs independent from the actual current state shall be delivered.

typecode:
typemask: The parameters typecode as well as typemask allow to use the DLU type code as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true:

DLUs type code AND typemask == typecode AND typemask

i.e. a typemask of zero will deliver all DLUs independent from their actual type code.

bufferSize: If the DLU info buffer is provided by the caller of the dvs_dlu_get_info_ext3() routine, the bufferSize parameter has to specify the size of this user provided buffer. If the DLU info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.

pCount: The parameter pCount holds the pointer to a user provided variable which will hold the actual number of retrieved DLUs after successful execution of the function.

ppBuffer: The parameter ppBuffer holds the pointer to a user provided variable which holds the pointer to the retrieved DLU info data buffer.
If the dvs_dlu_get_info_ext3() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppBuffer shall be initialized with the pointer to that buffer before the routine is called.
If the memory for the DLU info buffer shall be allocated dynamically by the interface itself, the ppBuffer variable shall be initialized by a NULL pointer before the routine is called. In the latter case, the ppBuffer variable will hold the pointer to the dynamically allocated buffer on successful return from dvs_dlu_get_info_ext3().

Return values:

DVS_IFC_OK: The request could be performed successfully.
The variable pointed to by ppBuffer holds the pointer to a data block, which holds all the retrieved DLU version information.

DVS_IFC_ERROR: Invalid parameter specified (pCount is a NULL pointer).

DVS_IFC_NOT_OPENED: The function could not be performed as the interface has not been opened before.

DVS_IFC_OUT_OF_MEMORY: The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.

DVS_IFC_BUFFER_OVFL: The static data buffer provided by the caller of the routine is not sufficient to hold the complete DLU info data structure.

DVS_IFC_NO_DLU: There are currently no DLUs available or the filter criteria are defined in a way so that no DLU passes the filter. In case of dynamic memory management, no data buffer is actually allocated and thus no memory buffer needs to be released.

Remark:

It should be kept in mind, that the required buffer size for given maximum number of DLU items is not the same as sizeof(DLU_PROPERTY_EXT3) * maximum number of DLUs. The reason for this is that the actual string data is not part of the DLU_PROPERTY_EXT3 structure. There is just a pointer to the string in. The actual string data needs to be copied behind the DLU info array, so that the required buffer size is typically much bigger than the pure array size. To get rid of the need for buffer size estimation on application side, the usage of dynamic buffer handling is recommended.

If dynamic buffer management is chosen (i.e. *ppBuffer is NULL and bufferSize is 0), the caller of the routine is responsible that the DLU data buffer is released by use of the dvs_release_dlu_info_buffer_ext3() function.

---

dvs_get_dlu_info_ext4( )

Synopsis

```
DVS_RESULT dvs_get_dlu_info_ext4(
    char                *dluname,
    UINT32              dluVersionVal,
    UINT32              dluVersionMask,
    UINT32              stateflags,
    UINT32              typecode,
    UINT32              typemask,
    UINT32              bufferSize,
    UINT32              *pCount,
    DLU_PROPERTY_EXT4   **ppBuffer)
```

Description

Purpose:

The dvs_get_dlu_info_ext4() function provides access to version information of all DLUs currently installed on target device.

The function may be used with static buffer provided by the caller of the routine as well as with dynamic buffer provided by the interface itself. In the latter case, the buffer provided by the dvs_get_dlu_info_ext4() routine must be freed explicitly by a call of the dvs_release_dlu_info_buffer_ext4().

Parameters:

dluname: The parameter dluname allows to restrict the search strategy of the dvs_get_dlu_info_ext4() routine to DLUs with the explicitly specified DLU name.
If the dluname parameter is an empty string or the dluname parameter is a NULL pointer, the name of the DLU is no filter criterion, i.e. dvs_get_dlu_info_ext4() will deliver the information of DLUs independent from their name.

dluVersionVal:
dluVersionMask: The parameters dluVersionVal as well as dluVersionMask allow using the DLU version as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true:

DLUs version AND dluVersionMask == dluVersionVal AND dluVersionMask

i.e. a dluVersionMask of zero will deliver all DLUs independent from their actual version.

stateflags: The stateflags parameter allows a restrictive query on DLUs within a specific state. The DVS_IFC_STATE_MASK_xxxx symbolic constants shall be used to define the expected DLU state filter criterion. The filter value is defined as bit field, i.e. an 'ored' combination of several state constants may be used.
The specific constant DVS_IFC_STATE_MASK_ALL shall be used, all DLUs independent from the actual current state shall be delivered.

typecode:
typemask: The parameters typecode as well as typemask allow to use the DLU type code as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true:

DLUs type code AND typemask == typecode AND typemask

i.e. a typemask of zero will deliver all DLUs independent from their actual type code.

bufferSize: If the DLU info buffer is provided by the caller of the dvs_dlu_get_info_ext4() routine, the bufferSize parameter has to specify the size of this user provided buffer. If the DLU info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.

pCount: The parameter pCount holds the pointer to a user provided variable which will hold the actual number of retrieved DLUs after successful execution of the function.

ppBuffer: The parameter ppBuffer holds the pointer to a user provided variable which holds the pointer to the retrieved DLU info data buffer.
If the dvs_dlu_get_info_ext4() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppBuffer shall be initialized with the pointer to that buffer before the routine is called.
If the memory for the DLU info buffer shall be allocated dynamically by the interface itself, the ppBuffer variable shall be initialized by a NULL pointer before the routine is called. In the latter case, the ppBuffer variable will hold the pointer to the dynamically allocated buffer on successful return from dvs_dlu_get_info_ext4().

Return values:

DVS_IFC_OK: The request could be performed successfully.
The variable pointed to by ppBuffer holds the pointer to a data block, which holds all the retrieved DLU version information.

DVS_IFC_ERROR: Invalid parameter specified (pCount is a NULL pointer).

DVS_IFC_NOT_OPENED: The function could not be performed as the interface has not been opened before.

DVS_IFC_OUT_OF_MEMORY: The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.

DVS_IFC_BUFFER_OVFL: The static data buffer provided by the caller of the routine is not sufficient to hold the complete DLU info data structure.

DVS_IFC_NO_DLU: There are currently no DLUs available or the filter criteria are defined in a way so that no DLU passes the filter. In case of dynamic memory management, no data buffer is actually allocated and thus no memory buffer needs to be released.

Remark:

It should be kept in mind, that the required buffer size for given maximum number of DLU items is not the same as sizeof(DLU_PROPERTY_EXT4) * maximum number of DLUs. The reason for this is that the actual string data is not part of the DLU_PROPERTY_EXT4 structure. There is just a pointer to the string in. The actual string data needs to be copied behind the DLU info array, so that the required buffer size is typically much bigger than the pure array size. To get rid of the need for buffer size estimation on application side, the usage of dynamic buffer handling is recommended.

If dynamic buffer management is chosen (i.e. *ppBuffer is NULL and bufferSize is 0), the caller of the routine is responsible that the DLU data buffer is released by use of the dvs_release_dlu_info_buffer_ext4() function.

---

dvs_get_dlu_info_ext5( )

Synopsis

```
DVS_RESULT dvs_get_dlu_info_ext5(
    char                *dluname,
    UINT32              dluVersionVal,
    UINT32              dluVersionMask,
    UINT32              stateflags,
    UINT32              typecode,
    UINT32              typemask,
    UINT32              bufferSize,
    UINT32              *pCount,
    DLU_PROPERTY_EXT5   **ppBuffer)
```

Description

Purpose:

The dvs_get_dlu_info_ext5() function provides access to version information of all DLUs currently installed on target device.

The function may be used with static buffer provided by the caller of the routine as well as with dynamic buffer provided by the interface itself. In the latter case, the buffer provided by the dvs_get_dlu_info_ext5() routine must be freed explicitly by a call of the dvs_release_dlu_info_buffer_ext5().

Parameters:

dluname: The parameter dluname allows to restrict the search strategy of the dvs_get_dlu_info_ext5() routine to DLUs with the explicitly specified DLU name.
If the dluname parameter is an empty string or the dluname parameter is a NULL pointer, the name of the DLU is no filter criterion, i.e. dvs_get_dlu_info_ext5() will deliver the information of DLUs independent from their name.

dluVersionVal:
dluVersionMask: The parameters dluVersionVal as well as dluVersionMask allow using the DLU version as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true:

DLUs version AND dluVersionMask == dluVersionVal AND dluVersionMask

i.e. a dluVersionMask of zero will deliver all DLUs independent from their actual version.

stateflags: The stateflags parameter allows a restrictive query on DLUs within a specific state. The DVS_IFC_STATE_MASK_xxxx symbolic constants shall be used to define the expected DLU state filter criterion. The filter value is defined as bit field, i.e. an 'ored' combination of several state constants may be used.
The specific constant DVS_IFC_STATE_MASK_ALL shall be used, all DLUs independent from the actual current state shall be delivered.

typecode:
typemask: The parameters typecode as well as typemask allow to use the DLU type code as filter criterion for retrieving DLU information. Information for a DLU is delivered as long as the following condition is true:

DLUs type code AND typemask == typecode AND typemask

i.e. a typemask of zero will deliver all DLUs independent from their actual type code.

bufferSize: If the DLU info buffer is provided by the caller of the dvs_dlu_get_info_ext5() routine, the bufferSize parameter has to specify the size of this user provided buffer. If the DLU info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.

pCount: The parameter pCount holds the pointer to a user provided variable which will hold the actual number of retrieved DLUs after successful execution of the function.

ppBuffer: The parameter ppBuffer holds the pointer to a user provided variable which holds the pointer to the retrieved DLU info data buffer.
If the dvs_dlu_get_info_ext4() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppBuffer shall be initialized with the pointer to that buffer before the routine is called.
If the memory for the DLU info buffer shall be allocated dynamically by the interface itself, the ppBuffer variable shall be initialized by a NULL pointer before the routine is called. In the latter case, the ppBuffer variable will hold the pointer to the dynamically allocated buffer on successful return from dvs_dlu_get_info_ext5().

Return values:

DVS_IFC_OK: The request could be performed successfully.
The variable pointed to by ppBuffer holds the pointer to a data block, which holds all the retrieved DLU version information.

DVS_IFC_ERROR: Invalid parameter specified (pCount is a NULL pointer).

DVS_IFC_NOT_OPENED: The function could not be performed as the interface has not been opened before.

DVS_IFC_OUT_OF_MEMORY: The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.

DVS_IFC_BUFFER_OVFL: The static data buffer provided by the caller of the routine is not sufficient to hold the complete DLU info data structure.

DVS_IFC_NO_DLU: There are currently no DLUs available or the filter criteria are defined in a way so that no DLU passes the filter. In case of dynamic memory management, no data buffer is actually allocated and thus no memory buffer needs to be released.

Remark:

It should be kept in mind, that the required buffer size for given maximum number of DLU items is not the same as sizeof(DLU_PROPERTY_EXT5) * maximum number of DLUs. The reason for this is that the actual string data is not part of the DLU_PROPERTY_EXT4 structure. There is just a pointer to the string in. The actual string data needs to be copied behind the DLU info array, so that the required buffer size is typically much bigger than the pure array size. To get rid of the need for buffer size estimation on application side, the usage of dynamic buffer handling is recommended.

If dynamic buffer management is chosen (i.e. *ppBuffer is NULL and bufferSize is 0), the caller of the routine is responsible that the DLU data buffer is released by use of the dvs_release_dlu_info_buffer_ext5() function.

---

dvs_get_hwinfo( )

Synopsis

```
DVS_RESULT dvs_get_hwinfo(
    UINT32          bufferSize,
    UINT32          *pActSize,
    char            **ppBuffer)
```

Description

Purpose:

The dvs_get_hwinfo() function provides access to hardware version information of all hardware components belonging to the target device in XML format according to the Unit schema.

The function may be used with static buffer provided by the caller of the routine as well as with dynamic buffer provided by the interface itself. In the latter case, the buffer provided by the dvs_get_hwinfo() routine must be freed explicitly by a call of the dvs_release_hwinfo_buffer().

Parameters:

bufferSize:	If the HW info buffer is provided by the caller of the dvs_get_hwinfo() routine, the bufferSize parameter has to specify the size of this user provided buffer. If the HW info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.
pActSize:	The parameter pActSize holds the pointer to a user provided variable which will hold the actual number of bytes delivered by the routine as hardware identification string. The value of the size variable includes the string termination zero character.
ppBuffer:	The parameter ppBuffer holds the pointer to a user provided variable which holds the pointer to the retrieved HW info data buffer. If the dvs_get_hwinfo() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppBuffer shall be initialized with the pointer to that buffer before the routine is called. If the memory for the HW info buffer shall be allocated dynamically by the interface itself, the ppBuffer variable shall be initialized by a NULL pointer before the routine is called. In the latter case, the ppBuffer variable will hold the pointer to the dynamically allocated buffer on successful return from dvs_get_hwinfo().

Return values:

DVS_IFC_OK:	The request could be performed successfully. The variable pointed to by ppBuffer holds the pointer to a data block which holds all the retrieved hardware information.
DVS_IFC_ERROR:	Invalid parameter specified (pActSize is a NULL pointer).
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_BUFFER_OVFL:	The static data buffer provided by the caller of the routine is not sufficient to hold the complete HW info data structure.
DVS_IFC_NO_HWINFO:	There is currently no HW information available. In case of dynamic memory management, no data buffer is actually allocated and thus no memory buffer need to be released.

Remark:

If dynamic buffer management is used (i.e. *ppBuffer is NULL), the caller of the routine is responsible that the HW info buffer is released by use of the dvs_release_hwinfo_buffer().

dvs_get_hwinfo_ext( )

Synopsis

```
DVS_RESULT dvs_get_hwinfo_ext(
    UINT32          bufferSize,
    UINT32          *pActSize,
    char            **ppBuffer)
```

Description

Purpose:

The dvs_get_hwinfo_ext() function provides access to hardware version information of all hardware components belonging to the target device in XML format according to the Unit2 schema.

The function may be used with static buffer provided by the caller of the routine as well as with dynamic buffer provided by the interface itself. In the latter case, the buffer provided by the dvs_get_hwinfo_ext() routine must be freed explicitly by a call of the dvs_release_hwinfo_buffer().

Parameters:

bufferSize:	If the HW info buffer is provided by the caller of the dvs_get_hwinfo_ext() routine, the bufferSize parameter has to specify the size of this user provided buffer. If the HW info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.
pActSize:	The parameter pActSize holds the pointer to a user provided variable which will hold the actual number of bytes delivered by the routine as hardware identification string. The value of the size variable includes the string termination zero character.
ppBuffer:	The parameter ppBuffer holds the pointer to a user provided variable which holds the pointer to the retrieved HW info data buffer. If the dvs_get_hwinfo_ext() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppBuffer shall be initialized with the pointer to that buffer before the routine is called. If the memory for the HW info buffer shall be allocated dynamically by the interface itself, the ppBuffer variable shall be initialized by a NULL pointer before the routine is called. In the latter case, the ppBuffer variable will hold the pointer to the dynamically allocated buffer on successful return from dvs_get_hwinfo_ext().

Return values:

DVS_IFC_OK:	The request could be performed successfully. The variable pointed to by ppBuffer holds the pointer to a data block which holds all the retrieved hardware information.
DVS_IFC_ERROR:	Invalid parameter specified (pActSize is a NULL pointer).
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_BUFFER_OVFL:	The static data buffer provided by the caller of the routine is not sufficient to hold the complete HW info data structure.
DVS_IFC_NO_HWINFO:	There is currently no HW information available. In case of dynamic memory management, no data buffer is actually allocated and thus no memory buffer need to be released.

Remark:

If dynamic buffer management is used (i.e. *ppBuffer is NULL), the caller of the routine is responsible that the HW info buffer is released by use of the dvs_release_hwinfo_buffer().

dvs_get_operation_mode( )

Synopsis

```
DVS_RESULT dvs_get_operation_mode(
    UINT32   *pOpm)
```

Description

Purpose:

The function dvs_get_operation_mode() evaluates the actual operational state of the target device.

Parameters:

pOpm:	The pointer to a variable which will hold the code for the operation mode which is currently relevant for the target device. For the evaluation of the operation mode, one of the DVS_IFC_OPM_xxx symbolic constants shall be used.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The variable pointed to by pOpm holds the code for the currently active operation mode.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_ERROR:	Invalid parameter specified (pOpm is a NULL pointer).

Remark:

None

---

dvs_get_swinfo( )

Synopsis

```
DVS_RESULT dvs_get_swinfo(
    UINT32         bufferSize,
    UINT32         *pActSize,
    char           **ppBuffer)
```

Description

Purpose:

The dvs_get_swinfo() function provides access to software version information in XML format according to the System schema.

Parameters:

bufferSize:	If the SW info buffer is provided by the caller of the dvs_get_swinfo() routine, the bufferSize parameter has to specify the size of this user provided buffer. If the SW info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.
pActSize:	The parameter pActSize holds the pointer to a user provided variable which will hold the actual number of bytes delivered by the routine as software identification string. The value of the size variable includes the string termination zero character.
ppBuffer:	The parameter ppBuffer holds the pointer to a user provided variable which holds the pointer to the retrieved SW info data buffer. If the dvs_get_swinfo() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppBuffer shall be initialized with the pointer to that buffer before the routine is called. If the memory for the SW info buffer shall be allocated dynamically by the interface itself, the ppBuffer variable shall be initialized by a NULL pointer before the routine is called. In the latter case, the ppBuffer variable will hold the pointer to the dynamically allocated buffer on successful return from dvs_get_swinfo().

Return values:

DVS_IFC_OK:	The request could be performed successfully. The variable pointed to by ppBuffer holds the pointer to a data block which holds all the retrieved software information.
DVS_IFC_ERROR:	Invalid parameter specified (pActSize is a NULL pointer).
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_BUFFER_OVFL:	The static data buffer provided by the caller of the routine is not sufficient to hold the complete software info data structure.
DVS_IFC_NO_SWINFO:	There is currently no SW information available. In case of dynamic memory management, no data buffer is actually allocated and thus no memory buffer need to be released.

Remark:

If dynamic buffer management is used (i.e. *ppBuffer is NULL), the caller of the routine is responsible that the software info buffer is released by use of the dvs_release_swinfo_buffer().

---

dvs_get_swinfo_ext( )

Synopsis

```
DVS_RESULT dvs_get_swinfo_ext(
    UINT32          bufferSize,
    UINT32          *pActSize,
    char            **ppBuffer)
```

Description

Purpose:

The dvs_get_swinfo_ext() function provides access to software version information in XML format according to the System2 schema.

Parameters:

bufferSize:	If the SW info buffer is provided by the caller of the dvs_get_swinfo_ext() routine, the bufferSize parameter has to specify the size of this user provided buffer. If the SW info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.
pActSize:	The parameter pActSize holds the pointer to a user provided variable which will hold the actual number of bytes delivered by the routine as software identification string. The value of the size variable includes the string termination zero character.
ppBuffer:	The parameter ppBuffer holds the pointer to a user provided variable which holds the pointer to the retrieved SW info data buffer. If the dvs_get_swinfo_ext() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppBuffer shall be initialized with the pointer to that buffer before the routine is called. If the memory for the SW info buffer shall be allocated dynamically by the interface itself, the ppBuffer variable shall be initialized by a NULL pointer before the routine is called. In the latter case, the ppBuffer variable will hold the pointer to the dynamically allocated buffer on successful return from dvs_get_swinfo_ext().

Return values:

DVS_IFC_OK:	The request could be performed successfully. The variable pointed to by ppBuffer holds the pointer to a data block which holds all the retrieved software information.
DVS_IFC_ERROR:	Invalid parameter specified (pActSize is a NULL pointer).
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_BUFFER_OVFL:	The static data buffer provided by the caller of the routine is not sufficient to hold the complete software info data structure.
DVS_IFC_NO_SWINFO:	There is currently no SW information available. In case of dynamic memory management, no data buffer is actually allocated and thus no memory buffer need to be released.

Remark:

If dynamic buffer management is used (i.e. *ppBuffer is NULL), the caller of the routine is responsible that the software info buffer is released by use of the dvs_release_swinfo_buffer().

---

dvs_get_ulu_info( )

Synopsis

```
DVS_RESULT dvs_get_ulu_info(
    UINT32          uluBaseAddress,
    UINT32          bufferSize,
    UINT32          *pCount,
    ULU_PROPERTY    **ppBuffer)
```

Description

Purpose:

The dvs_get_ulu_info() function provides access to version information of all ULUs currently installed on target device.

The function may be used with static buffer provided by the caller of the routine as well as with dynamic buffer provided by the interface itself. In the latter case, the buffer provided by the dvs_get_ulu_info() routine must be freed explicitly by a call of the dvs_release_ulu_info_buffer().

Parameters:

uluBaseAddress:	The parameter uluBaseAddress allows to restrict the ULU version retrieval to just one ULU on a well known base address. If the information of all ULUs shall be retrieved, the symbolic constant DVS_IPC_INVALID_ULU_ADDRESS shall be used as uluBaseAddress parameter.
bufferSize:	If the ULU info buffer is provided by the caller of the dvs_get_ulu_info() routine, the bufferSize parameter has to specify the size of this user provided buffer. If the ULU info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.
pCount:	The parameter pCount holds the pointer to a user provided variable which will hold the actual number of retrieved ULUs after successful execution of the function.
ppBuffer:	The parameter ppBuffer holds the pointer to a user provided variable which holds the pointer to the retrieved ULU info data buffer. If the dvs_get_ulu_info() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppBuffer shall be initialized with the pointer to that buffer before the routine is called. If the memory for the ULU info buffer shall be allocated dynamically by the interface itself, the ppBuffer variable shall be initialized by a NULL pointer before the routine is called. In the latter case, the ppBuffer variable will hold the pointer to the dynamically allocated buffer on successful return from dvs_get_ulu_info().

Return values:

DVS_IFC_OK:	The request could be performed successfully. The variable pointed to by ppBuffer holds the pointer to a data block which holds all the retrieved DLU version information.
DVS_IFC_ERROR:	Invalid parameter specified (pCount is a NULL pointer).
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_BUFFER_OVFL:	The static data buffer provided by the caller of the routine is not sufficient to hold the complete ULU info data structure.
DVS_IFC_NO_ULU:	There are currently no ULUs available or the filter criterion is defined in a way so that no ULU passes the filter. In case of dynamic memory management, no data buffer is actually allocated and thus no memory buffer need to be released.

Remark:

If dynamic buffer management is chosen (i.e. *ppBuffer is NULL and bufferSize is 0), the caller of the routine is responsible that the ULU data buffer is released by use of the dvs_release_ulu_info_buffer().

---

dvs_get_ulu_info_ext( )

Synopsis

```
DVS_RESULT dvs_get_ulu_info_ext(
    UINT32              uluBaseAddress,
    UINT32              bufferSize,
    UINT32              *pCount,
    ULU_PROPERTY_EXT    **ppBuffer)
```

Description

Purpose:

The dvs_get_ulu_info_ext() function provides access to version information of all ULUs currently installed on target device.

The function may be used with static buffer provided by the caller of the routine as well as with dynamic buffer provided by the interface itself. In the latter case, the buffer provided by the dvs_get_ulu_info() routine must be freed explicitly by a call of the dvs_release_ulu_info_buffer_ext().

Parameters:

uluBaseAddress:	The parameter uluBaseAddress allows to restrict the ULU version retrieval to just one ULU on a well known base address. If the information of all ULUs shall be retrieved, the symbolic constant DVS_IPC_INVALID_ULU_ADDRESS shall be used as uluBaseAddress parameter.
bufferSize:	If the ULU info buffer is provided by the caller of the dvs_get_ulu_info_ext() routine, the bufferSize parameter has to specify the size of this user provided buffer. If the ULU info buffer shall be provided by the DVS interface itself, the bufferSize parameter has to be set to zero.
pCount:	The parameter pCount holds the pointer to a user provided variable which will hold the actual number of retrieved ULUs after successful execution of the function.
ppBuffer:	The parameter ppBuffer holds the pointer to a user provided variable which holds the pointer to the retrieved ULU info data buffer. If the dvs_get_ulu_info_ext() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppBuffer shall be initialized with the pointer to that buffer before the routine is called. If the memory for the ULU info buffer shall be allocated dynamically by the interface itself, the ppBuffer variable shall be initialized by a NULL pointer before the routine is called. In the latter case, the ppBuffer variable will hold the pointer to the dynamically allocated buffer on successful return from dvs_get_ulu_info_ext().

Return values:

DVS_IFC_OK:	The request could be performed successfully. The variable pointed to by ppBuffer holds the pointer to a data block which holds all the retrieved DLU version information.
DVS_IFC_ERROR:	Invalid parameter specified (pCount is a NULL pointer).
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_BUFFER_OVFL:	The static data buffer provided by the caller of the routine is not sufficient to hold the complete ULU info data structure.
DVS_IFC_NO_ULU:	There are currently no ULUs available or the filter criterion is defined in a way so that no ULU passes the filter. In case of dynamic memory management, no data buffer is actually allocated and thus no memory buffer need to be released.

Remark:

If dynamic buffer management is chosen (i.e. *ppBuffer is NULL and bufferSize is 0), the caller of the routine is responsible that the ULU data buffer is released by use of the dvs_release_ulu_info_buffer_ext().

---

dvs_ifc_cleanup( )

Synopsis

```
DVS_RESULT dvs_ifc_cleanup(
    UINT32    clientHandle,
    UINT32    cleanupMode)
```

Description

Purpose:

The function dvs_ifc_cleanup() performs the cleanup request on target device. Cleanup is the removal of all (application-ware) software packages from target.

Parameters:

clientHandle:	The clientHandle as provided by the dvs_ifc_lock() request.
cleanupMode:	The cleanup operation may be requested in several modes. The following cleanup modes are supported: DVS_IFC_CLEANUP_MODE_APPLWARE: Only applicationware software packages are removed. DVS_IFC_CLEANUP_MODE_FULL: All software packages are removed. DVS_IFC_CLEANUP_MODE_GARBAGE: A garbage collection cleanup. All files related to DLUs which are no longer complete will be removed.

Return values:

DVS_IFC_OK:	The request could be performed successfully.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_INV_CLIENT_HANDLE:	The client handle is not valid. I.e. either that client handle has never been provided by DVS, or the client which once got that handle has already been released explicitly (by dvs_ifc_unlock()) or implicitly (by timeout condition).
DVS_IFC_ERROR:	Invalid or unexpected cleanup mode value.

Remark:

None

dvs_ifc_create_filesystem( )

Synopsis

```
DVS_RESULT dvs_ifc_create_filesystem(
    UINT32       clientHandle,
    const char   *parameter)
```

Description

Purpose:

The function dvs_ifc_create_filesystem() creates file system on target device.

Parameters:

clientHandle:	The clientHandle as provided by the dvs_ifc_lock() request.
parameter:	Optional parameter string for the create file system request.

Return values:

DVS_IFC_OK:	The request could be performed successfully.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_INV_CLIENT_HANDLE:	The client handle is not valid. I.e. either that client handle has never been provided by DVS, or the client which once got that handle has already been released explicitly (by dvs_ifc_unlock()) or implicitly (by timeout condition).

Remark:

None

dvs_ifc_dlu_add_metadata( )

Synopsis

```
DVS_RESULT dvs_ifc_dlu_add_metadata(
    UINT32          clientHandle,
    char            *dluName,
    UINT32          dluVersion,
    char            *paramId,
    char            *paramValue)
```

Description

Purpose:

The function dvs_ifc_dlu_add_metadata() writes meta data to a DLU. The function can only be performed if the DVS has been explicitly locked by a call to dvs_ifc_lock().

Parameters:

clientHandle:	The clientHandle as provided by the dvs_ifc_lock() request.
dluName:	The name of the DLU to be verified. The dluname parameter is case sensitive, i.e. the dluname must be specified exactly in the way which is used within the internal data structures of the DLU itself.
dluVersion:	The version of the DLU to add meta data to.
paramId:	The name of the meta data attribute to be set. Can be one of: Type, ProductID or Supplier.
paramValue:	The value to set for the attribute.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The meta data has been written to file.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_NO_FILE_SYSTEM:	The file system is not ready (not mounted).
DVS_IFC_INV_CLIENT_HANDLE:	One or more of the DLUs belonging to the referenced package are missing their data (probably the file has been removed).
DVS_IFC_DLU_BAD_CRC:	The client handle is not valid. I.e. either that client handle has never been provided by DVS, or the client which once got that handle has already been released explicitly (by dvs_ifc_unlock()) or implicitly (by timeout condition).
DVS_IFC_NO_DLU:	Could not find a DLU with the supplied name.
DVS_IFC_ERROR:	General error code. Can for example be returned if paramId does not specify any known attribute or if there is a problem reading the meta data file.

Remark:

None

dvs_ifc_dlu_close( )

Synopsis

```
DVS_RESULT dvs_ifc_dlu_close(
    UINT32              clientHandle,
    UINT32              dluFileHandle)
```

Description

Purpose:

The function dvs_ifc_dlu_close() closes a previously generated DLU. The function can only be performed, if DVS has been explicitly locked by the caller of the dvs_ifc_lock() request.

Parameters:

clientHandle:	The clientHandle as provided by the dvs_ifc_lock() request.
dluFileHandle:	The file handle as provided by the previously performed dvs_ifc_dlu_create() request.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The DLU payload file is closed and passed to the DVS DLU handling.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_INV_CLIENT_HANDLE:	The client handle is not valid. I.e. either that client handle has never been provided by DVS, or the client which once got that handle has already been released explicitly (by dvs_ifc_unlock()) or implicitly (by timeout condition).
DVS_IFC_INV_FILE_HANDLE:	The dluFileHandle parameter is invalid, i.e. that parameter does not reference a DLU previously created by the dvs_ifc_dlu_create() request.

Remark:

None

---

dvs_ifc_dlu_create( )

Synopsis

```
DVS_RESULT dvs_ifc_dlu_create(
    UINT32            clientHandle,
    TYPE_DLU_HEADER   *pDluHeader,
    UINT32            virtualAddress,
    UINT32            timeStamp,
    UINT32            permVal,
    UINT32            permMask,
    const char        *fileRef,
    const char        *hostFileName,
    UINT32            *pDluFileHandle)
```

Description

Purpose:

The function dvs_ifc_dlu_create() creates a new DLU. The function can only be performed if DVS has been explicitly locked by the caller of the dvs_ifc_dlu_create() request.

Parameters:

clientHandle:	The clientHandle as provided by the dvs_ifc_lock() request.
pDluHeader:	The DLU creation is based on Meta information provided within the so called DLU header. The parameter pDluHeader shall point to a client based DLU header structure, which holds the required information.
virtualAddress:	Although DVS is fully file system based and does not handle any kind of addresses for its DLU software packages, the DVS Interface allows to store a virtual address parameter as additional information within a DLU Meta record. That additional parameter serves as link between the file system based CC2 world and the address based world of the MVB download world. If the virtual address shall not be specified explicitly, the virtualAddress parameter shall hold the value DVS_IFC_VIRT_ADDR_UNKNOWN. In that case, DVS will internally assign a virtual address to the package.
timeStamp:	CC1 based DLUs provide a timeStamp as part of the DLU version information data. CC2 based DLUs do not provide that. But DVS interface provides the possibility to store that missing timeStamp information within the DLU Meta data record. If the time stamp shall not be used, a value of zero shall be used as time stamp parameter.
permVal:	On target platforms supporting file permission attributes, the permVal parameter together with the permMask parameter allows to specify the expected file permission of the payload file associated to the DLU. On platforms where the permission handling is not supported (e.g. CSS), the permVal as well as the permMask are not evaluated. Caller of that function on platforms not supporting file permission attributes shall use the value 0x00000000 as permVal as well as permMask.
permMask:	The permMask identifies which of the bits of the permVal shall actually be used as file permission attribute. A '1' bit within the permMask enables the usage of the corresponding bit within the permVal. A '0' bit inhibits the usage of the corresponding bit within the permVal. On platforms not supporting the permission attributes, the permMask parameter shall be set to 0x00000000.
fileRef:	The fileRef parameter allows the explicit specification of the DLU payload data file. If the name of the payload data file shall be assigned automatically by DVS, the fileRef parameter shall point to an empty string.
hostFileName:	CC1 based DLUs provide the hostFileName of the DLU as part of the DLU version information. CC2 based DLUs do not provide that. But DVS interface provides the possibility to store that missing hostFileName information within the DLU Meta data record. If the hostFileName shall not be used, the hostFileName parameter shall point to an empty string.
pDluFileHandle:	The pointer to a variable which will hold the DLU file handle on successful return from the function.

Return values:

DVS_IFC_OK:	The request could be performed successfully. A new DLU is created. The variable pointed to by the pDluFileHandle parameter will hold the DLU file handle after return from the function.
DVS_IFC_NOT_OPENED:	The function could not be performed, as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_INV_CLIENT_HANDLE:	The client handle is not valid. I.e. either that client handle has never been provided by DVS, or the client which once got that handle has already been released explicitly (by dvs_ifc_unlock()) or implicitly (by timeout condition).
DVS_IFC_FCREATE_FAILED:	The DLU create operation failed (e.g. if DVS currently has not write access to file system, hostFileName parameter refers to an invalid path).
DVS_IFC_DISK_FULL:	A DLU create operation failed because there is not enough disk (or memory) space on target device available.
DVS_IFC_ERROR:	Invalid parameter specified (pDluHeader or pDluFileHandle is a NULL pointer).

Remark:

None

---

dvs_ifc_dlu_create_ext( )

Synopsis

```
DVS_RESULT dvs_ifc_dlu_create_ext(
    UINT32            clientHandle,
    TYPE_DLU_HEADER   *pDluHeader,
    UINT32            virtualAddress,
    UINT32            timeStamp,
    UINT32            permVal,
    UINT32            permMask,
    const char        *fileRef,
    const char        *hostFileName,
    const char        *edPackName,
    UINT32            edPackVersion,
    const char        *sciName,
    UINT32            sciVersion,
    const char        *packageSource,
    UINT32            *pDluFileHandle)
```

Description

Purpose:

The function dvs_ifc_dlu_create_ext() creates a new DLU. The function can only be performed if DVS has been explicitly locked by the caller of the dvs_ifc_dlu_create_ext() request.

Parameters:

clientHandle: The clientHandle as provided by the dvs_ifc_lock() request.

pDluHeader: The DLU creation is based on Meta information provided within the so called DLU header. The parameter pDluHeader shall point to a client based DLU header structure, which holds the required information.

virtualAddress: Although DVS is fully file system based and does not handle any kind of addresses for its DLU software packages, the DVS Interface allows to store a virtual address parameter as additional information within a DLU Meta record. That additional parameter serves as link between the file system based CC2 world and the address based world of the MVB download world.
If the virtual address shall not be specified explicitly, the virtualAddress parameter shall hold the value DVS_IFC_VIRT_ADDR_UNKNOWN. In that case, DVS will internally assign a virtual address to the package.

timeStamp: CC1 based DLUs provide a timeStamp as part of the DLU version information data. CC2 based DLUs do not provide that. But DVS interface provides the possibility to store that missing timeStamp information within the DLU Meta data record.
If the time stamp shall not be used, a value of zero shall be used as time stamp parameter.

permVal: On target platforms supporting file permission attributes, the permVal parameter together with the permMask parameter allows to specify the expected file permission of the payload file associated to the DLU.
On platforms where the permission handling is not supported (e.g. CSS), the permVal as well as the permMask are not evaluated.
Caller of that function on platforms not supporting file permission attributes shall use the value 0x00000000 as permVal as well as permMask.

permMask: The permMask identifies which of the bits of the permVal shall actually be used as file permission attribute. A '1' bit within the permMask enables the usage of the corresponding bit within the permVal. A '0' bit inhibits the usage of the corresponding bit within the permVal.
On platforms not supporting the permission attributes, the permMask parameter shall be set to 0x00000000.

fileRef: The fileRef parameter allows the explicit specification of the DLU payload data file.
If the name of the payload data file shall be assigned automatically by DVS, the fileRef parameter shall point to an empty string.

hostFileName: CC1 based DLUs provide the hostFileName of the DLU as part of the DLU version information.
CC2 based DLUs do not provide that. But DVS interface provides the possibility to store that missing hostFileName information within the DLU Meta data record.
If the hostFileName shall not be used, the hostFileName parameter shall point to an empty string.

edPackName The name of the End Device Package, to which the DLU belongs to. The end device package name is a zero terminated ASCII string. The name is internally limited to 31 characters, i.e. edPackName strings with more than the maximum allowed string size or without missing string termination will be truncated to a length of 31 characters without any error indication.
If the edPackName shall not be used, the corresponding parameter shall point to an empty string.

edPackVersion The version of the End Device Package, to which the DLU belongs to. The version is encoded in standard Version/Release/Update/Evolution notation, whereas the most significant byte of the edPackVersion value holds the Version item and the least significant byte holds the Evolution item.
If the edPackName is not used (i.e. if edPackName holds an empty string), the edPackVersion shall be set to 0x00000000.

sciName The name of the Software Configuration Item, to which the DLU belongs to. The software configuration item name is a zero terminated ASCII string. The name is internally limited to 31 characters, i.e. edPackName strings with more than the maximum allowed string size or without missing string termination will be truncated to a length of 31 characters without any error indication.
If the sciName shall not be used, the corresponding parameter shall point to an empty string.

sciVersion	The version of the Software Configuration Item, to which the DLU belongs to. The version is encoded in standard Version/Release/Update/Evolution notation, whereas the most significant byte of the edPackVersion value holds the Version item and the least significant byte holds the Evolution item. If the sciName is not used (i.e. if sciName holds an empty string), the edPackVersion shall be set to 0x00000000.
packageSource	The origin of the software package, to which the DLU belongs to. The package source is specified as a two character zero terminated string. The packageSource string is typically 'BT' (for Bombardier software packages) or 'CU' (for customer specific software packaged). However, the DVS Interface will not perform any semantic check on the string itself. The packageSource string is internally limited to 2 characters, i.e. strings with more than the maximum allowed string size or without missing string termination will be truncated to a length of 2 characters without any error indication. If the packageSource shall not be used, the packageSource parameter shall point to an empty string.
pDluFileHandle:	The pointer to a variable which will hold the DLU file handle on successful return from the function.

Return values:

DVS_IFC_OK:	The request could be performed successfully. A new DLU is created. The variable pointed to by the pDluFileHandle parameter will hold the DLU file handle after return from the function.
DVS_IFC_NOT_OPENED:	The function could not be performed, as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_INV_CLIENT_HANDLE:	The client handle is not valid. I.e. either that client handle has never been provided by DVS, or the client which once got that handle has already been released explicitly (by dvs_ifc_unlock()) or implicitly (by timeout condition).
DVS_IFC_FCREATE_FAILED:	The DLU create operation failed (e.g. if DVS currently has not write access to file system, hostFileName parameter refers to an invalid path).
DVS_IFC_DISK_FULL:	A DLU create operation failed because there is not enough disk (or memory) space on target device available.
DVS_IFC_ERROR:	Invalid parameter specified (pDluHeader or pDluFileHandle is a NULL pointer).

Remark:

None

---

dvs_ifc_dlu_create_ext2( )

Synopsis

```
DVS_RESULT dvs_ifc_dlu_create_ext2(
    UINT32            clientHandle,
    TYPE_DLU_HEADER   *pDluHeader,
    UINT32            virtualAddress,
    UINT32            timeStamp,
    UINT32            permVal,
    UINT32            permMask,
    const char        *fileRef,
    const char        *hostFileName,
    const char        *edPackName,
    UINT32            edPackVersion,
    const char        *sciName,
    UINT32            sciVersion,
```

```
const char        *packageSource,
UINT32            *pDluFileHandle,
UINT32            edCrc,
const char        *edInfo,
const char        *edTimeStamp,
UINT32            sciCrc,
const char        *sciInfo,
const char        *sciTimeStamp)
```

Description

Purpose:

The function dvs_ifc_dlu_create_ext2() creates a new DLU. The function can only be performed if DVS has been explicitly locked by the caller of the dvs_ifc_dlu_create_ext2() request.

Parameters:

clientHandle: The clientHandle as provided by the dvs_ifc_lock() request.

pDluHeader: The DLU creation is based on Meta information provided within the so called DLU header. The parameter pDluHeader shall point to a client based DLU header structure, which holds the required information.

virtualAddress: Although DVS is fully file system based and does not handle any kind of addresses for its DLU software packages, the DVS Interface allows to store a virtual address parameter as additional information within a DLU Meta record. That additional parameter serves as link between the file system based CC2 world and the address based world of the MVB download world.
If the virtual address shall not be specified explicitly, the virtualAddress parameter shall hold the value DVS_IFC_VIRT_ADDR_UNKNOWN. In that case, DVS will internally assign a virtual address to the package.

timeStamp: CC1 based DLUs provide a timeStamp as part of the DLU version information data. CC2 based DLUs do not provide that. But DVS interface provides the possibility to store that missing timeStamp information within the DLU Meta data record.
If the time stamp shall not be used, a value of zero shall be used as time stamp parameter.

permVal: On target platforms supporting file permission attributes, the permVal parameter together with the permMask parameter allows to specify the expected file permission of the payload file associated to the DLU.
On platforms where the permission handling is not supported (e.g. CSS), the permVal as well as the permMask are not evaluated.
Caller of that function on platforms not supporting file permission attributes shall use the value 0x00000000 as permVal as well as permMask.

permMask: The permMask identifies which of the bits of the permVal shall actually be used as file permission attribute. A '1' bit within the permMask enables the usage of the corresponding bit within the permVal. A '0' bit inhibits the usage of the corresponding bit within the permVal.
On platforms not supporting the permission attributes, the permMask parameter shall be set to 0x00000000.

fileRef: The fileRef parameter allows the explicit specification of the DLU payload data file.
If the name of the payload data file shall be assigned automatically by DVS, the fileRef parameter shall point to an empty string.

hostFileName: CC1 based DLUs provide the hostFileName of the DLU as part of the DLU version information.
CC2 based DLUs do not provide that. But DVS interface provides the possibility to store that missing hostFileName information within the DLU Meta data record.
If the hostFileName shall not be used, the hostFileName parameter shall point to an empty string.

edPackName The name of the End Device Package, to which the DLU belongs to. The end device package name is a zero terminated ASCII string. The name is internally limited to 31 characters, i.e. edPackName strings with more than the maximum allowed string size or without missing string termination will be truncated to a length of 31 characters without any error indication.
If the edPackName shall not be used, the corresponding parameter shall point to an empty string.

edPackVersion The version of the End Device Package, to which the DLU belongs to. The version is encoded in standard Version/Release/Update/Evolution notation, whereas the most significant byte of the edPackVersion value holds the Version item and the least significant byte holds the Evolution item.
If the edPackName is not used (i.e. if edPackName holds an empty string), the edPackVersion shall be set to 0x00000000.

sciName The name of the Software Configuration Item, to which the DLU belongs to. The software configuration item name is a zero terminated ASCII string. The name is internally limited to 31 characters, i.e. edPackName strings with more than the maximum allowed string size or without missing string termination will be truncated to a length of 31 characters without any error indication.
If the sciName shall not be used, the corresponding parameter shall point to an empty string.

sciVersion	The version of the Software Configuration Item, to which the DLU belongs to. The version is encoded in standard Version/Release/Update/Evolution notation, whereas the most significant byte of the edPackVersion value holds the Version item and the least significant byte holds the Evolution item. If the sciName is not used (i.e. if sciName holds an empty string), the edPackVersion shall be set to 0x00000000.
packageSource	The origin of the software package, to which the DLU belongs to. The package source is specified as a two character zero terminated string. The packageSource string is typically 'BT' (for Bombardier software packages) or 'CU' (for customer specific software packaged). However, the DVS Interface will not perform any semantic check on the string itself. The packageSource string is internally limited to 2 characters, i.e. strings with more than the maximum allowed string size or without missing string termination will be truncated to a length of 2 characters without any error indication. If the packageSource shall not be used, the packageSource parameter shall point to an empty string.
pDluFileHandle:	The pointer to a variable which will hold the DLU file handle on successful return from the function.
edCrc:	The CRC32 checksum of the End Device Package, to which the DLU belongs.
edInfo:	Info string of up to 31 characters for the End Device Package, to which the DLU belongs.
edTimeStamp:	Creation time of the End Device Package, to which the DLU belongs.
sciCrc:	The CRC32 checksum of the Software Configuration Item, to which the DLU belongs.
sciInfo:	Info string of up to 31 characters for the Software Configuration Item, to which the DLU belongs.
sciTimeStamp:	Creation time of the Software Configuration Item, to which the DLU belongs.

Return values:

DVS_IFC_OK:	The request could be performed successfully. A new DLU is created. The variable pointed to by the pDluFileHandle parameter will hold the DLU file handle after return from the function.
DVS_IFC_NOT_OPENED:	The function could not be performed, as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_INV_CLIENT_HANDLE:	The client handle is not valid. I.e. either that client handle has never been provided by DVS, or the client which once got that handle has already been released explicitly (by dvs_ifc_unlock()) or implicitly (by timeout condition).
DVS_IFC_FCREATE_FAILED:	The DLU create operation failed (e.g. if DVS currently has not write access to file system, hostFileName parameter refers to an invalid path).
DVS_IFC_DISK_FULL:	A DLU create operation failed because there is not enough disk (or memory) space on target device available.
DVS_IFC_ERROR:	Invalid parameter specified (pDluHeader or pDluFileHandle is a NULL pointer).

Remark:

None

dvs_ifc_dlu_write_data( )

Synopsis

```
DVS_RESULT dvs_ifc_dlu_write_data(
    UINT32            clientHandle,
    UINT32            dluFileHandle,
    UINT32            fileOffset,
    UINT32            byteCount,
```

```
    const char        *pData)
```

Description

Purpose:

The function dvs_ifc_dlu_write_data() writes payload data to a DLU, previously generated by a dvs_ifc_dlu_create() request. The function can only be performed if DVS has been explicitly locked by the caller of the dvs_ifc_dlu_write_data() request.

Parameters:

clientHandle:	The clientHandle as provided by the dvs_ifc_lock() request.
dluFileHandle:	The file handle as provided by the previously performed dvs_ifc_dlu_create() request.
fileOffset:	The file offset, where the payload data shall be written to.
byteCount:	The number of bytes to be written to the payload data file.
pData:	Pointer to the data block, which shall be written to payload file.

Return values:

DVS_IFC_OK:	The request could be performed successfully. All data could be written successfully to payload file.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_INV_CLIENT_HANDLE:	The client handle is not valid. I.e. either that client handle has never been provided by DVS, or the client which once got that handle has already been released explicitly (by dvs_ifc_unlock()) or implicitly (by timeout condition).
DVS_IFC_INV_FILE_HANDLE:	The dluFileHandle parameter is invalid, i.e. that parameter does not reference a DLU previously created by the dvs_ifc_dlu_create() request.
DVS_IFC_DISK_FULL:	The DLU write operation failed because there is not enough disk (or memory) space on target device available.
DVS_IFC_WR_SEQ_ERROR:	An out of sequence DLU write operation has been requested, but that is not supported. DLU write operations must be performed without any gaps, i.e. each consecutive DLU write operation must continue at the current 'end of file' position, which results from the last write operation.

Remark:

None

---

dvs_ifc_lock( )

Synopsis

```
DVS_RESULT dvs_ifc_lock(
    UINT32          timeoutInSeconds,
    UINT32          lockMode,
    const char      *clientInfoString,
    UINT32          *pClientHandle)
```

Description

Purpose:

The function dvs_ifc_lock() requests exclusive access to the DVS internal data structures. All functionality, which might change the state of the data structures managed by DVS is only accepted if DVS has been successfully locked for the requesting client software.

Parameters:

timeoutInSeconds:	In order to prevent from infinitively locked DVS (e.g. in case of client software malfunction if unlock is not performed), the lock functionality provides a time controlled automatic unlock mechanism. The timeout is specified in seconds. If no request associated to the returned client handle takes place within the specified time span, the lock is automatically released.
lockMode:	The lockMode parameter is reserved for future extensions. Currently lockMode is not evaluated by DVS. Unless specified otherwise, the lockMode parameter shall be set to zero.
clientInfoString:	The clientInfoString is an optional parameter. It should intentionally hold a human readable string which identifies the current owner of the lock. That string is used by external interfaces (e.g. MCPM) to indicate the reason for failed lock requests to the requester of the lock.
pClientHandle:	The pointer to a variable which will hold the client handle on successful return from the function.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The caller of the function successfully locked DVS. The variable pointed to by pClientHandle holds the client handle value, which has to be used for further requests.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_ERROR:	Invalid parameter specified (pClientHandle is a NULL pointer).
DVS_IFC_ALREADY_LOCKED:	The lock request failed as DVS is already locked by another client software.

Remark:

None

dvs_ifc_server_is_local( )

Synopsis

```
DVS_RESULT dvs_ifc_server_is_local(
    void)
```

Description

Purpose:

The routine dvs_ifc_server_is_local() checks whether the DVS Interface Server currently in use is on the same device as the client.

Parameters: None

Return values:

DVS_IFC_OK:	The request could be performed successfully. The DVS Interface Server currently in use is actually on the same machine as the client which requests the function.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_ERROR:	The DVS Interface Server currently in use is actually not on the same machine as the client which requests the function.

Remark:

None

dvs_ifc_unlock( )

Synopsis

```
DVS_RESULT dvs_ifc_unlock(
```

```
    UINT32    clientHandle)
```

Description

Purpose:

The function dvs_ifc_unlock() releases a client handle, which has been requested previously by a dvs_ifc_lock() request.

Parameters:

clientHandle: The clientHandle as provided by the dvs_ifc_lock() request.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The client handle is released and DVS is no longer locked for the calling client software.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this, the function could not be performed successfully.
DVS_IFC_INV_CLIENT_HANDLE:	The client handle is not valid. I.e. either that client handle has never been provided by DVS, or the client which once got that handle has already been released explicitly (by dvs_ifc_unlock()) or implicitly (by timeout condition).

Remark:

None

dvs_indicate_cleanup( )

Synopsis

```
DVS_RESULT dvs_indicate_cleanup(
    void)
```

Description

Purpose:

If a cleanup indication is explicitly assigned by a call of dvs_asign_cleanup(), the routine dvs_indicate_cleanup() is called by the DVS server before a cleanup operation within operation mode OSRUN is performed.

If the indication is actually assigned, the cleanup operation will only be performed, if the indicated client application confirms the cleanup by a corresponding return value of the dvs_indicate_cleanup() function.

Parameters: None

Return values:

DVS_IFC_OK:	The cleanup request is explicitly granted by the application. The cleanup operation can be performed.
DVS_IFC_DONT_CLEANUP:	The cleanup request is not granted by the application connected by DVS interface. Either the application explicitly denies the cleanup, or there is no application which has assigned a callback handler for that request. In both cases the DVS server shall not perform the cleanup operation.

Remark:

None

dvs_indicate_dlu_state( )

Synopsis

```
DVS_RESULT dvs_indicate_dlu_state(
    char      *dluname,
```

```
    UINT32    dluversion,
    UINT32    dlustate)
```

Description

Purpose:

Within DVS each DLU is assigned to a discrete dedicated state at each time during life time of the DLU (i.e. a DLU may be considered as a finite state machine). The dvs_indicate_dlu_state() function provides means for manipulating this state. For a modification of the DLU state, the DLU must be specified explicitly by its unique identification which consists of the DLU name and the DLU version.

Parameters:

dluname:	The name of the DLU for which the status shall be changed. The dluname parameter is case sensitive, i.e. the dluname must be specified exactly in the way which is used within the internal data structures of the DLU itself.
dluversion:	The version of the DLU for which the status shall be changed.
dlustate:	The expected state transition. Use one of the DVS_IFC_STATE_xxx symbolic constants for the specification of the expected state transition.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The state of the DLU could be changed as requested.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_INV_TRANSITION:	The expected state transition is not allowed within the current state of the DLU.
DVS_IFC_NO_DLU:	The parameter dluname together with the parameter dluversion do not reference a DLU currently existing on the device.

Remark:

Not all state transitions are allowed within all states of a DLU. Besides this not all types of DLUs do support the same state transitions. Not supported state transitions are answered by a corresponding error code.

---

dvs_mcpm_request( )

Synopsis

```
DVS_RESULT dvs_mcpm_request(
    UINT32    requestSize,
    char      *pRequestData,
    UINT32    *pReplySize,
    char      **ppReplyBuffer)
```

Description

Purpose:

A DVS server typically consists of an MCP/L2 server as well as of an MCPM server. Whereas the MCP/L2 server is part of the (NRTOS or CSS) operating system, the MCPM server is unfortunately an external component which is not linked together with the DVS core itself.

For consistency reasons, the complete DVS handling must be localized within one and only one software instance on every target. This is why there is a need for a handover of MCPM requests to the DVS server via an external communication channel. The DVS interface is used for this purpose.

The function dvs_mcpm_request() provides the means for performing those MCPM requests 'remote' via the DVS interface communication channel.

The MCPM handler itself just handles the MCPM communication part and forwards any incoming MCPM request to the DVS server. The DVS server evaluates and performs the request and sends the respond back via DVS interface to the MCPM handler which than finalizes the request by sending back the response to the originator.

The function dvs_mcpm_request() can be used with dynamic as well as with static reply buffer handling. If the variable pointed to by pReplySize is set to zero before calling the function, the data buffer needed for the reply message will be allocated internally. In that case the allocated buffer must later on explicitly freed by the caller of the function by calling the dvs_release_mcpm_reply_buffer() function. If the variable holds a value not equal zero, the reply buffer has to be provided by the caller of the function. The variable shall indicate the size of the provided buffer.

Parameters:

requestSize:	The overall size of the MCPM request which have to be performed by DVS server.
pRequestData:	A pointer to the data buffer which holds the MCPM request data in raw format (i.e. in the same binary stream format as it has been received from network). The MCPM handler will not perform any endianness corrections on those data, i.e. the full handling of the package is done within DVS server.
pReplySize:	The parameter pReplySize points to a variable which holds the size of the reply data buffer. For dynamic reply data buffer handling, that variable shall be set to zero before calling the dvs_mcpm_request() function. For static reply data buffer handling, that variable shall represent the actual size of the data buffer provided by the caller of the routine. After successful handling of the MCPM package, the variable pointed to by pReplySize will hold the actual size of the reply package to be sent back by the MCPM handler.
ppReplyBuffer:	The parameter ppReplyBuffer holds the pointer to an user supplied variable which holds the pointer to the MCPM reply buffer. If the dvs_mcpm_request() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppReplyBuffer shall be initialized with the pointer to that buffer before the routine is called. If the memory for the reply buffer shall be allocated dynamically by the interface itself, the ppReplyBuffer variable shall be initialized with a NULL pointer before the routine is called. In the latter case, the ppReplyBuffer variable will hold the pointer to the dynamically allocated buffer.

Return values:

DVS_IFC_OK:	The request could be performed successfully.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_BUFFER_OVFL:	The static data buffer provided by the caller of the routine is not sufficient to hold the complete MCPM reply.

Remark:

None

dvs_mcpm_request2( )

Synopsis

```
DVS_RESULT dvs_mcpm_request2(
    UINT32    clientIp,
    UINT32    requestSize,
    char      *pRequestData,
    UINT32    *pReplySize,
    char      **ppReplyBuffer)
```

Description

Purpose:

A DVS server typically consists of an MCP/L2 server as well as of an MCPM server. Whereas the MCP/L2 server is part of the (NRTOS or CSS) operating system, the MCPM server is unfortunately an external component which is not linked together with the DVS core itself.

For consistency reasons, the complete DVS handling must be localized within one and only one software instance on every

target. This is why there is a need for a handover of MCPM requests to the DVS server via an external communication channel. The DVS interface is used for this purpose.

The function dvs_mcpm_request2() provides the means for performing those MCPM requests 'remote' via the DVS interface communication channel.

The MCPM handler itself just handles the MCPM communication part and forwards any incoming MCPM request to the DVS server. The DVS server evaluates and performs the request and sends the respond back via DVS interface to the MCPM handler which than finalizes the request by sending back the response to the originator.

The function dvs_mcpm_request2() can be used with dynamic as well as with static reply buffer handling. If the variable pointed to by pReplySize is set to zero before calling the function, the data buffer needed for the reply message will be allocated internally. In that case the allocated buffer must later on explicitly freed by the caller of the function by calling the dvs_release_mcpm_reply_buffer() function. If the variable holds a value not equal zero, the reply buffer has to be provided by the caller of the function. The variable shall indicate the size of the provided buffer.

Parameters:

clientIp:	The IP address of the client which originated the MCPM request. The IP address is encoded as 32 bit numerical value in network order (as it's also used within most operating systems socket implementations).
requestSize:	The overall size of the MCPM request which have to be performed by DVS server.
pRequestData:	A pointer to the data buffer which holds the MCPM request data in raw format (i.e. in the same binary stream format as it has been received from network). The MCPM handler will not perform any endianness corrections on those data, i.e. the full handling of the package is done within DVS server.
pReplySize:	The parameter pReplySize points to a variable which holds the size of the reply data buffer. For dynamic reply data buffer handling, that variable shall be set to zero before calling the dvs_mcpm_request2() function. For static reply data buffer handling, that variable shall represent the actual size of the data buffer provided by the caller of the routine. After successful handling of the MCPM package, the variable pointed to by pReplySize will hold the actual size of the reply package to be sent back by the MCPM handler.
ppReplyBuffer:	The parameter ppReplyBuffer holds the pointer to an user supplied variable which holds the pointer to the MCPM reply buffer. If the dvs_mcpm_request2() function shall be used with static memory buffers (i.e. a buffer provided by the caller of the function), the parameter ppReplyBuffer shall be initialized with the pointer to that buffer before the routine is called. If the memory for the reply buffer shall be allocated dynamically by the interface itself, the ppReplyBuffer variable shall be initialized with a NULL pointer before the routine is called. In the latter case, the ppReplyBuffer variable will hold the pointer to the dynamically allocated buffer.

Return values:

DVS_IFC_OK:	The request could be performed successfully.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_BUFFER_OVFL:	The static data buffer provided by the caller of the routine is not sufficient to hold the complete MCPM reply.

Remark:

None

---

dvs_open( )

Synopsis

```
DVS_RESULT dvs_open(
    char   *targetAddr)
```

Description

Purpose:

For implementations where the DVS interface is implemented by a TCP/IP communication channel, a call of the dvs_open() routine is mandatory. All other functions provided by this interface won't work without calling the dvs_open() in advance.

For implementations where the DVS interface is used internally without any communication channel inbetween DVS interface server and DVS interface client, a call of dvs_open() is not mandatory.

Nevertheless the routine dvs_open() is provided for those implementations too, so that applications can call the routine independent from the actual implementation.

Parameters:

targetAddr	The IP address of the target on which the DVS interface server shall be connected. The address is specified as standard dotted IP address string (e.g. "192.168.0.2"). If the targetAddr points to an empty string, or to the predefined string "localhost" or targetAddr is a NULL pointer, the DVS interface server on the local device is contacted. For implementations where the DVS interface is not provided by a TCP/IP channel, the parameter targetAddr is disregarded.

Return values:

DVS_IFC_OK:	The DVS interface could be initialized successfully.
DVS_IFC_ALREADY_OPENED:	The DVS interface has already been opened before. No action has been performed.
DVS_IFC_NO_SERVER:	There is no DVS interface server available.
DVS_IFC_CONNECT_FAILED, DVS_IFC_SOCKET_ERROR, DVS_IFC_NO_SOCKET:	Due to some internal communication errors there is currently no link to the DVS interface server available.

Remark:

None

---

dvs_ping( )

Synopsis

```
DVS_RESULT dvs_ping(
    UINT32   *pDvsServerVersion,
    UINT32   *pDvsFlags )
```

Description

Purpose:

The dvs_ping() function provides a simple communication check between DVS interface server and DVS interface client.

Parameters:

pDvsServerVersion:	The parameter pDvsServerVersion holds the pointer to a user provided variable which will hold the version key of the actual DVS server implementation.
pDvsFlags:	The parameter pDvsServerVersion holds the pointer to a user provided variable which will hold the flags of the actual DVS server implementation. Use one of the DVS_IFC_FLAG_xxxx symbolic constants for the evaluation of the variable.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The variable pointed to by pDvsServerVersion holds the version key of the actual DVS server implementation.
DVS_IFC_ERROR:	Invalid parameter specified (pDvsServerVersion is a NULL pointer).
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.

Remark:

If the DVS interface implementation does not use any communication channel inbetween server and client, the dvs_ping() routine just delivers the DVS version without any communication checks.

dvs_release_dlu_info_buffer( )

Synopsis

```
void dvs_release_dlu_info_buffer(
    DLU_PROPERTY   *pBuffer)
```

Description

Purpose:

Depending on the parameterization, the dvs_get_dlu_info() function, dynamically allocates a buffer and returns that buffer to the caller of this routine. This dynamically allocated buffer must be explicitly released by use of the dvs_release_dlu_info_buffer().

Parameters:

pBuffer:	The pointer to the data buffer provided by the dvs_get_dlu_info() function.

Return values:

None

Remark:

None

dvs_release_dlu_info_buffer_ext( )

Synopsis

```
void dvs_release_dlu_info_buffer_ext(
    DLU_PROPERTY_EXT   *pBuffer)
```

Description

Purpose:

Depending on the parameterization, the dvs_get_dlu_info_ext() function, dynamically allocates a buffer and returns that buffer to the caller of this routine. This dynamically allocated buffer must be explicitly released by use of the dvs_release_dlu_info_buffer_ex

Parameters:

pBuffer:	The pointer to the data buffer provided by the dvs_get_dlu_info_ext() function.

Return values:

None

Remark:

None

dvs_release_dlu_info_buffer_ext2( )

Synopsis

```
void dvs_release_dlu_info_buffer_ext2(
```

```
    DLU_PROPERTY_EXT2    *pBuffer)
```

Description

Purpose:

Depending on the parameterization, the dvs_get_dlu_info_ext2() function, dynamically allocates a buffer and returns that buffer to the caller of this routine. This dynamically allocated buffer must be explicitly released by use of the dvs_release_dlu_info_buffer_ex

Parameters:

pBuffer: The pointer to the data buffer provided by the dvs_get_dlu_info_ext2() function.

Return values:

None

Remark:

None

---

dvs_release_dlu_info_buffer_ext3( )

Synopsis

```
void dvs_release_dlu_info_buffer_ext3(
    DLU_PROPERTY_EXT3    *pBuffer)
```

Description

Purpose:

Depending on the parameterization, the dvs_get_dlu_info_ext3() function, dynamically allocates a buffer and returns that buffer to the caller of this routine. This dynamically allocated buffer must be explicitly released by use of the dvs_release_dlu_info_buffer_ex

Parameters:

pBuffer: The pointer to the data buffer provided by the dvs_get_dlu_info_ext3() function.

Return values:

None

Remark:

None

---

dvs_release_dlu_info_buffer_ext4( )

Synopsis

```
void dvs_release_dlu_info_buffer_ext4(
    DLU_PROPERTY_EXT4    *pBuffer)
```

Description

Purpose:

Depending on the parameterization, the dvs_get_dlu_info_ext4() function, dynamically allocates a buffer and returns that buffer to the caller of this routine. This dynamically allocated buffer must be explicitly released by use of the dvs_release_dlu_info_buffer_ex

Parameters:

pBuffer: The pointer to the data buffer provided by the dvs_get_dlu_info_ext4() function.

Return values:

None

Remark:

None

---

dvs_release_dlu_info_buffer_ext5( )

Synopsis

```
void dvs_release_dlu_info_buffer_ext5(
    DLU_PROPERTY_EXT5   *pBuffer)
```

Description

Purpose:

Depending on the parameterization, the dvs_get_dlu_info_ext5() function, dynamically allocates a buffer and returns that buffer to the caller of this routine. This dynamically allocated buffer must be explicitly released by use of the dvs_release_dlu_info_buffer_ex

Parameters:

pBuffer: The pointer to the data buffer provided by the dvs_get_dlu_info_ext5() function.

Return values:

None

Remark:

None

dvs_release_hwinfo_buffer( )

Synopsis

```
void dvs_release_hwinfo_buffer(
    char   *pBuffer)
```

Description

Purpose:

Depending on the parameterization, the dvs_get_hwinfo() function, dynamically allocates a buffer and returns that buffer to the caller of this routine. This dynamically allocated buffer must be explicitly released by use of the dvs_release_hwinfo_buffer().

Parameters:

pBuffer: The pointer to the data buffer provided by the dvs_get_hwinfo() function.

Return values:

None

Remark:

None

dvs_release_mcpm_reply_buffer( )

Synopsis

```
void dvs_release_mcpm_reply_buffer(
    char   *pBuffer)
```

Description

Purpose:

Depending on the parameterization, the dvs_mcpm_request() function, dynamically allocates a buffer and returns that buffer to the caller of this routine. This dynamically allocated buffer must be explicitly released by use of the dvs_release_mcpm_reply_buffer().

Parameters:

pBuffer: The pointer to the data buffer provided by the dvs_mcpm_request() function.

Return values:

None

Remark:

None

dvs_release_swinfo_buffer( )

Synopsis

```
void dvs_release_swinfo_buffer(
    char     *pBuffer)
```

Description

Purpose:

Depending on the parameterization, the dvs_get_swinfo() function, dynamically allocates a buffer and returns that buffer to the caller of this routine. This dynamically allocated buffer must be explicitly released by use of the dvs_release_swinfo_buffer().

Parameters:

pBuffer: The pointer to the data buffer provided by the dvs_get_swinfo() function.

Return values:

None

Remark:

None

---

dvs_release_ulu_info_buffer( )

Synopsis

```
void dvs_release_ulu_info_buffer(
    ULU_PROPERTY   *pBuffer)
```

Description

Purpose:

Depending on the parameterization, the dvs_get_ulu_info() function, dynamically allocates a buffer and returns that buffer to the caller of this routine. This dynamically allocated buffer must be explicitly released by use of the dvs_release_ulu_info_buffer().

Parameters:

pBuffer: The pointer to the data buffer provided by the dvs_get_ulu_info() function.

Return values:

None

Remark:

None

---

dvs_release_ulu_info_buffer_ext( )

Synopsis

```
void dvs_release_ulu_info_buffer_ext(
    ULU_PROPERTY_EXT   *pBuffer)
```

Description

Purpose:

Depending on the parameterization, the dvs_get_ulu_info_ext() function, dynamically allocates a buffer and returns that buffer to the caller of this routine. This dynamically allocated buffer must be explicitly released by use of the dvs_release_ulu_info_buffer_ex

Parameters:

pBuffer: The pointer to the data buffer provided by the dvs_get_ulu_info_ext() function.

Return values:

None

Remark:

None

dvs_set_operation_mode( )

Synopsis

```
DVS_RESULT dvs_set_operation_mode(
    UINT32   newOpm,
    UINT32   *pOldOpm)
```

Description

Purpose:

The function dvs_set_operation_mode() modifies the actual operational state of the target device.

Parameters:

newOpm:	The expected operational mode. The parameter shall hold one of the DVS_IFC_OPM_xxx symbolic constants.
pOldOpm:	The pointer to a variable which will hold the code for the operation mode which before the requested transition. For the evaluation of the operation mode, one of the DVS_IFC_OPM_xxx symbolic constants shall be used.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The variable pointed to by pOldOpm holds the code for the currently active operation mode.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_ERROR:	Invalid parameter specified (pOldOpm is a NULL pointer or newOpm is no valid operation mode).
DVS_IFC_INV_TRANSITION:	The current operation mode of the operating system does not allow any modification of the operation mode via DVS interface.

Remark:

None

dvs_verify_dlu( )

Synopsis

```
DVS_RESULT dvs_verify_dlu(
    char      *dluname,
    UINT32    dluversion)
```

Description

Purpose:

The function dvs_verify_dlu() provides means for verification and validation of DLU stored on target device. The verification is based on CRC calculation and the comparison of the calculated CRC with the CRC stored within meta data of the DLU.

The DLU to be verified must be specified explicitly by its unique identification which consists of the DLU name and the DLU version.

Parameters:

dluname:	The name of the DLU to be verified. The dluname parameter is case sensitive, i.e. the dluname must be specified exactly in the way which is used within the internal data structures of the DLU itself.
dluversion:	The version of the DLU to be verified.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The verification of the DLU indicates no problem, i.e. the CRC stored within meta information fits to the actually calculated CRC.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_NO_DLU:	The parameter dluname together with the parameter dluversion do not reference a DLU currently existing on the device.
DVS_IFC_DLU_NODATA:	The parameters reference a DLU which consists of meta information, but the actual DLU application data is missing (probably the corresponding file has been removed).
DVS_IFC_DLU_BAD_CRC:	The CRC verification detects some error within the DLU, i.e. some parts of the DLU or parts of the meta information file are corrupted or somebody has replaced data file without updating meta information.

Remark:

None

---

dvs_verify_edsp( )

Synopsis

```
DVS_RESULT dvs_verify_edsp(
    const char     *edspName,
    UINT8          calcDluCrc,
    UINT32         *crc)
```

Description

Purpose:

The function dvs_verify_edsp() provides means for verification and validation of an EDSP stored on target device. The verification is based on CRC calculation of all contained SCIs, the EDSP name, the EDSP version, the EDSP info field and the EDSP creation date and the comparison of the calculated CRC with the CRC stored for the EDSP. The EDSP to be verified must be specified explicitly by its unique identification which is the EDSP name.

Parameters:

edspName:	The name of the EDSP to be verified. The edspName parameter is case sensitive, i.e. the edspName must be specified exactly in the same way as it is used within the internal data structures of the EDSP itself.
calcDluCrc:	If TRUE, the CRCs of all contained DLUs are recalculated from the data on disk. Otherwise the stored CRCs are just summed up.
crc:	The calculated CRC of the EDSP.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The verification of the EDSP indicates no problem, i.e. the CRC stored within the meta data file matches the calculated CRC.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_NO_DLU:	The parameter edspName does not reference any EDSP currently installed on the device.
DVS_IFC_DLU_NODATA:	One or more of the DLUs belonging to the referenced package are missing their data (probably the file has been removed).
DVS_IFC_DLU_BAD_CRC:	The CRC verification detects some error within the EDSP, i.e. some parts of a DLU, SCI or parts of a meta information file are corrupted or somebody has replaced a data file without updating the meta information.

Remark:

None

---

dvs_verify_sci( )

Synopsis

```
DVS_RESULT dvs_verify_sci(
    const char     *edspName,
    const char     *sciName,
    UINT8          calcDluCrc,
    UINT32         *crc)
```

Description

Purpose:

The function dvs_verify_sci() provides means for verification and validation of an SCI stored on the target device. The verification is based on CRC calculation of all contained DLUs, the SCI name, the SCI version, the SCI info field and the SCI creation date and the comparison of the calculated CRC with the CRC stored for the SCI. The SCI to be verified must be specified explicitly by its unique identification which is the SCI name and the name of the containing EDSP.

Parameters:

edspName:	The name of the EDSP the SCI to be verified belongs to. The edspName parameter is case sensitive, i.e. the edspName must be specified exactly in the same way as it is used within the internal data structures of the EDSP itself.
sciName:	The name of the SCI to be verified. The sciName parameter is case sensitive, i.e. the sciName must be specified exactly in the same way as it is used within the internal data structures of the SCI itself.
calcDluCrc:	If TRUE, the CRCs of all contained DLUs are recalculated from the data on disk. Otherwise the stored CRCs are just summed up.
crc:	The calculated CRC of the SCI.

Return values:

DVS_IFC_OK:	The request could be performed successfully. The verification of the SCI indicates no problem, i.e. the CRC stored within the meta data file matches the calculated CRC.
DVS_IFC_NOT_OPENED:	The function could not be performed as the interface has not been opened before.
DVS_IFC_OUT_OF_MEMORY:	The internal communication stack failed on allocation of dynamic memory. Due to this the function could not be performed successfully.
DVS_IFC_NO_DLU:	The parameter sciName together with the parameter edspName does not reference any SCI currently installed on the device.
DVS_IFC_DLU_NODATA:	One or more of the DLUs belonging to the referenced package are missing their data (probably the file has been removed).
DVS_IFC_DLU_BAD_CRC:	The CRC verification detects some error within the SCI, i.e. some parts of a DLU or parts of a meta information file are corrupted or somebody has replaced a data file without updating the meta information.

Remark:

None

2.9 EH - Exception Handler Service

The Exception Handling service ensures a controlled behavior in case of exceptions and normal device shutdowns.

The standard exception handling sequence is:
- Stop the periodic scheduler.
- Create the exception record in NVRAM.
- Move the exception record to a file.
- Run all application shutdown functions.
- Wait for 2000ms to let the application tasks finish what they do.
- Reset the device.

Normal device shutdowns are just like exception shutdowns, but no exception record is created.

It is possible to connect one shutdown function to each application. All the shutdown functions are called before the device is reset.

Note: The periodic scheduler is stopped before the shutdown functions are called. This means that all currently executing periodic tasks may run to completion, but no periodic task will be re-scheduled to run.
After the shutdown functions have run the non-periodic application tasks run for another 2000ms before the device resets.

There are several kinds of exceptions, the most common ones are bus error and program error.
If a task causes an exception it is suspended and the CSS exception handler routine will be called.
The exception handler issues an SE-Log telling which task and application caused the exception.
A file is created containing information about the latest exception on the device.
The exception log files are stored in the /data0/log/exc directory and are time-stamped for easy reference.
The 5 newest log files are saved for future reference.

In case of an exception in the RTS monitor, e.g. a command accessing the memory, the exception will be caught by the monitor itself and the monitor task, but not the device, is restarted.

---

eh_assert( )

Synopsis

```
void eh_assert(
    const char   *msg)                  /* In: The assertion failed message */
```

Description

This function prints the assertion failed message and restarts the device.
Runs the actual assert handling in the context of the exception task.

---

eh_connect( )

Synopsis

```
INT16 eh_connect(
    const char *appl_name,          /* In: Application name */
    FUNCPTR     function,           /* In: The function that is called when the
                                       exception is raised */
    UINT8       priority)           /* In: The priority of the application when
                                       shutting down the device */
```

Description

This function connects an application to the exception handler.

In case of an exception causing the device to shut down the functions will be called in order of their priority.

Priority 0 is highest and 255 is lowest, and in case of the same priority the application first calling this function will get its shutdown function called first.

Maximum 30 shutdown functions can be added.

If an application already have added a shutdown function, a new call to this function will replace the old function reference and priority with this new one.

Returns OK on success or ERROR in the following cases:

- Any parameter in the call is invalid
- The application does not exist
- There are no more room for shutdown functions

2.10 FM - File Management Service

The File Manager (FM) service:

The FM service provides API functions for creating and deleting RAM disks and for monitoring the free space of a file system. In addition it provides a function to retrieve the name of a file system.

---

fm_format_file_system( )

Synopsis

```
int fm_format_file_system(
    UINT32 fs_num,                              /* In: The FS number */
    UINT32 fs_flags,                            /* In: Various options */
    UINT32 boot_image_size)                     /* In: Reserved size of FS */
```

Description

This function erases a file system and then re-formats it.

The boot_image_size argument is deprecated and is not evaluated.

No fs_flags are currently supported.

Returns OK if the file system could be erased and formatted.
Returns ERROR if:
- The file system does not exist.
- The file system is not a supported file system.
- The format operation was interrupted by the p_show_progress function.
- The erase operation itself failed.
- The format operation itself failed.
- A format may already be in progress.

---

fm_get_fs_name( )

Synopsis

```
int fm_get_fs_name(
    UINT32     fs_num,                  /* In:  The FS number */
    size_t     fs_name_len,             /* In:  Length of fs_name buffer */
    char       fs_name[],               /* Out: The FS name */
    BOOL      *p_is_mounted)            /* Out: TRUE if FS is mounted */
```

Description

Given the file system number return the name of the file system in a user-supplied buffer.

Returns OK if fs_num is a valid local file system index.
Returns ERROR if:
- The indexed file system does not exist.
- The argument fs_name is NULL.
- The argument fs_name_len is 0 (zero).
- fs_name is too small to contain the name of the file system.

---

fm_get_fs_space( )

Synopsis

```
int fm_get_fs_space(
    const char *fs_name,                 /* In:  The FS name */
    UINT32     *p_total_kb,              /* Out: Total device size in KB */
    UINT32     *p_free_kb )              /* Out: Free space in KB */
```

Description

This function is used to find out how big a file system is and how much of it is free.

The results are always given in units of kbytes (1024 bytes).
One or both out-pointers may be NULL if the information is not interesting.

Returns OK if the file system information could be retrieved.
Returns ERROR if:
- File system does not exist.
- File system information could not be retrieved.

fm_part_and_format_sata( )

Synopsis

```
STATUS fm_part_and_format_sata(
        void)
```

Description

Creates and formats the two partitions /app0/ and /data0/ without user interaction during its process.

To avoid conditional compilation in user code (e.g. DVS) the name is kept with _sata even though EOS doesn't have a SATA disk.

fm_ram_disk_create( )

Synopsis

```
int fm_ram_disk_create(
    UINT32      size_kb,                    /* In:  The requested size in KB */
    UINT32     *p_fs_num )                  /* Out: File system number */
```

Description

Create a RAM disk and format it with a HRFS file system.
CSS allows a maximum of 5 RAM disks to be created at any time.
Disks will be named "/fmram0/" .. "/fmram4/".
Applications should use API function fm_get_fs_name to get the name of the created RAM disk device.

Returns OK if the RAM disk was successfully created and formatted.
Returns ERROR if a RAM disk could not be created or if it could not be formatted or if the arguments are bad.

fm_ram_disk_delete( )

Synopsis

```
int fm_ram_disk_delete(
    UINT32 fs_num )                                 /* In:  File system number */
```

Description

Delete an existing RAM disk.

Returns OK if the RAM disk was deleted.
Returns ERROR if the RAM disk does not exist or it could not be deleted.

fm_tar_extract( )

Synopsis

```
int fm_tar_extract(
    const char *file_name )                  /* In: TAR file to extract from */
```

Description

Extracts a tar archive to the current directory.

Returns OK if successful.
Returns ERROR if:
- File does not exist.
- An error occurred while extracting file(s) from the tar file.

---

fm_tar_extract_to_dir( )

Synopsis

```
int fm_tar_extract_to_dir(
    const char *file_name,                /* In: The TAR file to extract */
    const char *root_name )               /* In: Existing DIR to extract to */
```

Description

Extracts a tar archive to a specific directory.
The directory must already exist, but may be empty.
If the TAR file contains a directory hierarchy it will be created in the specified directory.
Files with same name in the dir will be silently overwritten.

Returns OK if successful.
Returns ERROR if:
- File does not exist.
- Directory does not exist.
- An error occurred while extracting file(s) from the tar file.

---

fm_tar_extract_to_dir_with_cb( )

Synopsis

```
int fm_tar_extract_to_dir_with_cb(
    const char   *file_name,          /* In: The TAR file to extract */
    const char   *root_name,          /* In: Existing DIR to extract to */
    p_tar_cb_ftn  p_cb,               /* In: The call-back function pointer */
    int           cb_context )        /* In: Argument to call-back */
```

Description

Extracts a tar archive to a specific directory.
The directory must already exist, but may be empty.
If the TAR file contains a directory hierarchy it will be created in the specified directory.
Files with same name in the dir will be silently overwritten.
If a call-back function is supplied it will be called when a file has been extracted.

Returns OK if successful.
Returns ERROR if:
- File does not exist.
- Directory does not exist.
- An error occurred while extracting file(s) from the tar file.

2.11 FTPC - File Transfer Protocol Client Service

This service provides an FTP Client API to put and get files on a remote FTP Server.

---

ftpc_file_get( )

Synopsis

```
INT16 ftpc_file_get(
    char* server,           /* In: name of server host */
    char* user,             /* In: user name for server login */
    char* passwd,           /* In: password for server login */
    char* acct,             /* In: account for server login. Typically "". */
    char* dirname,          /* In: directory to cd to before getting file */
    char* remote_file,      /* In: filename to get from server */
    const char* local_file, /* In: filename of local file to copy to */
    INT16* p_status)        /* Out: Status on the operation */
```

Description

This routine transfers a file from a remote FTP server to the local file system. It performs the following actions:

1. Establish a connection to the FTP server on the specified host.
2. Log in with the specified user name, password, and account, as necessary for the particular host.
3. Set the transfer type to image by sending the command ”TYPE I”.
4. Change to the specified directory by sending the command ”CWD dirname”.
5. Send the command ”RETR specified filename as an argument, and establishes a data connection.
6. Transfer the file from the server to the local file system.
7. Close the connection and returns the result of the transfer.

Returns	***p_status**	**Description**
OK	OK	Successful in getting file
ERROR	FTPC_FILE_CREATE_FAILED	Could not create local file
ERROR	FTPC_CONNECT_FAILED	Could not connect to server. Wrong
-	-	user, password, account, directory
-	-	or filename
ERROR	FTPC_DISK_FULL	Could not store complete file because
-	-	local disk full
ERROR	FTPC_READ_FAILED	Could not read complete file on server
ERROR	FTPC_XFER_NOT_COMPLETE	Could not complete FTP session
ERROR	FTPC_QUIT_FAILED	Could not quit FTP session

---

ftpc_file_put( )

Synopsis

```
INT16 ftpc_file_put(
    char* server,           /* In: name of server host */
    char* user,             /* In: user name for server login */
    char* passwd,           /* In: password for server login */
    char* acct,             /* In: account for server login. Typically "". */
    char* dirname,          /* In: directory to cd to before storing file */
    char* remote_file,      /* In: filename to put on server */
    const char* local_file, /* In: filename of local file to copy to server */
    INT16* p_status)        /* Out: Status on the operation */
```

Description

This routine transfers a file from the local file system to a remote FTP server. It performs the following actions:

1. Establish a connection to the FTP server on the specified host.
2. Log in with the specified user name, password, and account, as necessary for the particular host.
3. Set the transfer type to image by sending the command ”TYPE I”.
4. Change to the specified directory by sending the command ”CWD dirname”.
5. Send the command STOR specified filename as an argument, and establishes a data connection.
6. Transfer the file to the server.
7. Close the connection and returns the result of the transfer.

Returns	***p_status**	**Description**
OK	OK	Successful in storing file
ERROR	FTPC_FILE_OPEN_FAILED	Could not open local file
ERROR	FTPC_CONNECT_FAILED	Could not connect to server. Wrong user,
-	-	password, account, directory or filename
ERROR	FTPC_DISK_FULL	Could not store complete file because
-	-	server disk full
ERROR	FTPC_WRITE_FAILED	Could not write complete file to server
ERROR	FTPC_XFER_NOT_COMPLETE	Could not complete FTP session
ERROR	FTPC_QUIT_FAILED	Could not quit FTP session

ftpc_set_port( )

Synopsis

```
INT16 ftpc_set_port(
    unsigned short port_no)  /* In: Server host FTP-port number */
```

Description

This routine can be used for specifying the FTP-port number to use for subsequent communication with a FTP-server. As every FTP-operation (file put or get) creates a new connection to a FTP-server, this routine can be called prior any file put or get operation.

The default value of the FTP-port is 21.

Returns	**Description**
OK	FTP-port successfully changed

2.12 HR - Hardware Resources Service

The CSS HR service is a common interface to access hardware resources in a device.

The provided API can be used on all devices supported by CSS and in case the hardware does not exist on the device the function will return ERROR and the status parameter will reflect the reason why it fails.

The reason why the hardware does not exist can be that the device is never equipped with that kind of hardware or is not configured with the hardware.

The reason for returning ERROR can also be that the hardware is present but defect.

Interrupt Handling:

Interrupt handling in firmware (outside of CSS) is possible for specific interrupt sources.

An application may assign an interrupt handler function to an interrupt source. Those interrupts may then be enabled and disabled by the application.

Valid enum values for interrupt source (IRQs) are:

IRQ name	Device(s)	IRQ Source
HR_IRQ_POWER_FAIL	VCU-C2/EOS	Power fail shutdown
HR_IRQ_MVB_0	VCU-C2/EOS	MVB bus 1 IRQ0
HR_IRQ_MVB_1	VCU-C2/EOS	MVB bus 1 IRQ1
HR_IRQ_MVB2_0	VCU-C2/EOS	MVB bus 2 IRQ0
HR_IRQ_MVB2_1	VCU-C2/EOS	MVB bus 2 IRQ1

The following enum values have been kept in CSS4 for compatibility with CSS3, but are not used. The reason is to avoid breaking the integral constants in enumeration HR_IRQ_SRC.

IRQ name	Device(s)	IRQ Source
HR_IRQ_DIPF_GDU	DCU2	DIPF GDU power fail
HR_IRQ_IPAC_B0	VCUC	IPAC B interrupt 0
HR_IRQ_IPAC_B1	VCUC	IPAC B interrupt 1
HR_IRQ_STACK_BOARD1	VCUC	Stack board 1
HR_IRQ_STACK_BOARD2	VCUC	Stack board 2
HR_IRQ_STACK_BOARD3	VCUC	Stack board 3
HR_IRQ_STACK_BOARD4	VCUC	Stack board 4
HR_IRQ_STACK_BOARD5	VCUC	Stack board 5
HR_IRQ_STACK_BOARD6	VCUC	Stack board 6
HR_IRQ_IPAC_A0	VCUC/DCU2	Alias for HR_IRQ_MVB2_0
HR_IRQ_IPAC_A1	VCUC/DCU2	Alias for HR_IRQ_MVB2_1

Timestamp Facility:

The HR service provides a high-resolution time-stamping facility.
The API for the HR time-stamping facility are:
hr_timestamp_enable, hr_timestamp and hr_timestamp_disable.

Note 1:	The need to enable and disable the timestamp timer effectively makes this facility non-re-entrant. CSS does not check for reentrancy issues.
Note 2:	The CSS system time (available through the os_c_get) has even higher resolution, is always available and does not have to be enabled/disabled to use.

Serial EEPROM access:

The HR service provides access to EEPROM areas in the HW.
Several EEPROM areas are supported, depending on the specific device type.

The data is not read from the physical EEPROM device due to time performance, instead a shadow area is used.
The shadow area is updated during startup. Different devices are equipped with different number and size of EEPROMs.

The external EEPROM on the VCU-C2 (MOBAD) does not have a shadow area. It is read directly from the physical memory.

To select which EEPROM to access use the following defines:

Define	Value	Description
EEPROM_ICM_CPU	0	CPU board EEPROM for VCU-C2 and EOS
EEPROM_EXT	1	External EEPROM for VCU-C2 i.e. MOBAD
EEPROM_ICM_BASE	3	Base I/O board EEPROM for VCU-C2
EEPROM_2_CPU	4	CPU board 2:nd EEPROM for VCU-C2
EEPROM_ICM_EXP1	6	Expansion board 1 EEPROM for VCU-C2
EEPROM_ICM_EXP2	7	Expansion board 2 EEPROM for VCU-C2
EEPROM_ICM_EXP3	8	Expansion board 3 EEPROM for VCU-C2
EEPROM_ICM_EXP4	9	Expansion board 4 EEPROM for VCU-C2
EEPROM_ICM_EXP5	10	Expansion board 5 EEPROM for VCU-C2
EEPROM_ICM_EXP6	11	Expansion board 6 EEPROM for VCU-C2
EEPROM_ICM_STACK1	12	Stack board 1 EEPROM for VCU-C2
EEPROM_ICM_STACK2	13	Stack board 2 EEPROM for VCU-C2
EEPROM_ICM_STACK3	14	Stack board 3 EEPROM for VCU-C2
EEPROM_ICM_STACK4	15	Stack board 4 EEPROM for VCU-C2
EEPROM_ICM_STACK5	16	Stack board 5 EEPROM for VCU-C2
EEPROM_ICM_STACK6	17	Stack board 6 EEPROM for VCU-C2

Device Condition:

The HR service provides an interface to query for the temperature and battery status, when supported, of a device.

hr_temp_status_get	provides the temperature from internal sensors with a resolution of 1 degree C.
hr_temp_100_status_get	provide the temperature from internal sensors with a resolution of 1/100 degree C. The actual physical temp sensor may have lower resolution.
hr_batt_status_get	provides the status of the battery in the device. If a battery is not available or not supported then this is noted in the return code.
hr_batt_use	is maintained for backwards compatibility. It has no other function than to affect the status argument of hr_batt_status_get.

Several functions are provided, for legacy reasons, that return information about specific memory areas in the device.

The Hardware Watchdog:

All devices running CSS include a HW watchdog. It is used to detect when a device no longer works correctly.
If the watchdog is not triggered correctly it resets the device without warning. CSS detects this condition on the next restart and logs an entry in the SE-log.

Error modes include (among others):
- Memory has been corrupted (overwritten).
- HW interrupts overwhelming the OS.
- A high-priority task does not yield to allow low-priority tasks to run.

This watchdog is normally handled entirely by CSS by a low priority task. If, e.g., an application exerts a very high load on the CPU and does not allow the internal watchdog task to run then the HW watchdog will reset the device.

An application may take over the triggering of the watchdog, e.g., during a very CPU-intensive operation. To do so the application must first call hr_wdog_appl once, then regularly (at least every 2.5 seconds) call hr_wdog_trigger to keep the watchdog from resetting the device.
When the application has finished its CPU-intensive operation it may put CSS in charge of triggering the watchdog by calling hr_wdog_css.

Cache and Memory Helpers:

When writing device drivers for HW that is currently unsupported by CSS the following functions may be needed:

hr_memory_barrier
hr_cache_flush
hr_cache_invalidate

Using these functions may be required for proper interfacing to HW, but excessive use of these functions may seriously impair performance of the calling code.

Device Status LEDs:

For EOS, all LEDs are controlled entirely by the HW. Because of this, it is not possible to control any LED from CSS or the application on an EOS device.

A VCU-C2 provides status information via the PWR, HW, SW and ERR LEDs.
The PWR and HW LEDs are controlled entirely by the HW.

The SW and ERR LEDs are controlled by a combination of SW and HW.
Due to HW design it is not possible to turn both the SW and the ERR LEDs off. If that is attempted then the ERR LED will be lit anyway.

Normally, after the device starts, the status LEDs are controlled by CSS and used to signal the CSS mode:

CSS Mode	VCU-C2 LEDs
IDLE	SW off ERR on
RUN	SW on, ERR off

An application may request control over the status LEDs by calling hr_led_appl. In that case CSS saves its current state and refrains from modifying the LEDs.
The application may use hr_led_set to control the status LEDs until the application returns the control to CSS by calling hr_led_css.
When CSS regains control over the status LEDs it restores the saved state to the LEDs.

User Designated LEDs:

LED marked L1 on the front plate of the VCU-C2 is user-defined by default.
An application may directly call hr_led_set to modify the settings of the user-designated LED.

Communication Activity LEDs:

All VCU-C2 devices have LEDs marked MVB1, MVB2, ETH1, ETH2, USB1, USB2, SERA, SERB and MON. These are used to signal communication activity for the resp. interface.

An application may use function hr_led_comm_act_set to switch between showing communication activity and having control of a LED status. See API hr_set_comm_act_set for details.

---

hr_batt_status_get( )

Synopsis

```
INT16 hr_batt_status_get(
    UINT16  *p_voltage,          /* Out: Battery voltage [mV] */
    INT16   *p_status)           /* Out: result status */
```

Description

Check the battery status and return battery voltage and status.

Returns the following value and status code:

Return value	*p_status	Description
OK	OK	Battery OK
ERROR	HR_BATT_DEAD	Battery dead - replace now
ERROR	HR_BATT_NOT_SUPRV	No battery supervision
ERROR	HR_BATT_NOT_EXISTS	No battery exists
ERROR	ERROR	Battery can not be read

Note: No battery exists in VCU-C2 and EOS. Function kept for backward compatibility. Function will always return HR_BATT_NOT_EXISTS and 0V.

---

hr_batt_use( )

Synopsis

```
INT16 hr_batt_use(
    BOOL   flag,                 /* In:  Turn on/off battery supervision */
    INT16  *p_status)            /* Out: result status */
```

Description

This function can be used to turn on/off the battery check in the function hr_batt_status_get.

Returns the following value and status code:

Return value	*p_status	Description
ERROR	NOT_SUPPORTED	No battery exists
ERROR	ERROR	Wrong parameter in the call
OK	OK	Success

Note: No battery exists in VCU-C2 or EOS. Function kept for backward compatibility. Function will always return OK (Success).

hr_cache_flush( )

Synopsis

```
INT32 hr_cache_flush(
    void    *p_start,                   /* In:  The start address */
    size_t  len)                        /* In:  The area size */
```

Description

This function is only intended to be used when writing device drivers or other HW-related code.

A cache-flush operation may be needed if an output buffer resides in cacheable memory and the HW that reads the output buffer does so without involving the main CPU, e.g. using another CPU or a DMA (direct memory access) engine.

Using this function may be required for proper interfacing to HW, but excessive use of this function may seriously impair performance of the calling code.

This function internally ensures that the cache flush operation has completed in the HW before returning.

Returns the result of the cache flush operation.

hr_cache_invalidate( )

Synopsis

```
INT32 hr_cache_invalidate(
    void    *p_start,                   /* In:  The start address */
    size_t  len)                        /* In:  The area size */
```

Description

This function is only intended to be used when writing device drivers or other HW-related code.

A cache-invalidate operation may be needed if an input buffer resides in cacheable memory and the HW that writes to the input buffer does so without involving the main CPU, e.g. using another CPU or a DMA (direct memory access) engine.

Using this function may be required for proper interfacing to HW, but excessive use of this function may seriously impair performance of the calling code.

This function internally ensures that the cache flush operation has completed in the HW before returning.

Returns the result of the cache invalidate operation.

hr_crc_check( )

Synopsis

```
INT16 hr_crc_check(
    INT16  *p_status)    /* Out: result status */
```

Description

Checks non application specific CRCs on the device, e.g. the MOBAD/12SX02 on the VCU-C2. Each device specific crc check will generate an SE-Log if it fails.

Returns the following value and status code:

Return value	*p_status	Description
OK	OK	All CRCs OK
ERROR	HR_CRC_NOT_IMPLEMENTED	Not implemented for this device
ERROR	HR_CRC_MOBAD_BAD	Bad MOBAD CRC do not trust device
ERROR	ERROR	Could not check CRC due to H/W error

hr_ee_read( )

Synopsis

```
INT16 hr_ee_read(
    UINT16  id,                 /* In:  EEPROM identification number */
    UINT16  address,            /* In:  start address in EEPROM */
    UINT16  length,             /* In:  length of the data */
    BYTE    *p_data,            /* Out: pointer to buffer where the data is
                                        stored */
    INT16   *p_status)          /* Out: result status */
```

Description

This function reads data from a serial EEPROM.

Returns the following value and status codes:

Return value	*p_status	Description
ERROR	WRONG_PAR_VALUE	Wrong value on parameter in the call
-	-	or EEPROM does not exist
ERROR	ERROR	The EEPROM can not be read
OK	OK	Success

hr_ee_write( )

Synopsis

```
INT16 hr_ee_write(
    UINT16       id,            /* In:  EEPROM identification number */
    UINT16       address,       /* In:  start address in EEPROM */
    UINT16       length,        /* In:  length of the data */
    const BYTE   *p_data,       /* In:  pointer to buffer containing the data
                                        to be written */
    INT16        *p_status)     /* Out: result status */
```

Description

Writes data of an external or internal serial EEPROM.

When the external EEPROM on the VCU-C2 (MOBAD) is selected the data is written to the physical memory directly and the checksum is updated.

Returns the following value and status code:

Return value	*p_status	Description
ERROR	WRONG_PAR_VALUE	Wrong value on parameter in the call
-	-	or EEPROM does not exist
ERROR	ERROR	The EEPROM can not be written
ERROR	NOT_SUPPORTED	For EOS
OK	OK	Success

hr_flash_addr_size_get( )

Synopsis

```
INT16 hr_flash_addr_size_get(
    void            **pp_addr,  /* Out: flash memory base address */
    unsigned long   *p_size,    /* Out: flash memory size */
    INT16           *p_status)  /* Out: result status */
```

Description

Get the base address and size for the flash memory.

Returns OK and *p_status is set to OK always.

hr_get_controller_id( )

Synopsis

```
STATUS hr_get_controller_id (
```

```
        UINT8   *p_value)     /* Out: Controller ID */
```

Description

This function gets the Controller ID.

The Controller ID is an identifier which is defined by the backplane connector. Returns OK if information is available, otherwise ERROR

Note: This function is not supported in CCU-O2 and will return ERROR if used.

hr_get_netconfig_default( )

Synopsis

```
BOOL hr_get_netconfig_default (
        void)
```

Description

This function finds out if the default network configuration is used or not.

Returns TRUE if network configuration is setup by using e.g. ipaddr, gatewayip etc.
Returns FALSE if network configuration is setup by another construction e.g. a lookup table.

Note: This function is not supported in CCU-O2 and will return TRUE if used.

hr_info_get( )

Synopsis

```
INT16 hr_info_get(
    UINT16            list_index,       /* In:  defines instance to return */
    OS_HW_INFO_BOARD  *p_hw_info_board, /* Out: the buffer for the requested
                                              board instance */
    UINT16            *p_nr_hw_info,    /* Out: number of board entries
                                              available */
    INT16             *p_status)        /* Out: result status */
```

Description

CSS maintains an internal database of the HW detected during device startup.
This database may be queried by the application. I.e., to provide HW revision information.
The number of available hardware information entries is returned in the output parameter p_nr_hw_info.
The user may use list_index=0 to query the number of available entries.

Returns the following value and status code:

Return value	*p_status	Description
ERROR	WRONG_PAR_VALUE	Wrong value on parameter in the call
OK	OK	Success or number of available entries=0

hr_irq_connect( )

Synopsis

```
INT16 hr_irq_connect(
    HR_IRQ_SRC      irq,                    /* In: The interrupt source */
    HR_IRQ_FUNC     isr_routine,            /* In: The handler function */
    int             isr_arg)                /* In: Argument to function */
```

Description

This function is only intended to be used when writing device drivers or other HW-related code.

Connect a function to handle a specific interrupt.

Returns OK if the irq handler is connected.
Returns ERROR if:
- The irq argument is not valid.
- isr_routine is NULL.
- The irq is already connected.

---

hr_irq_disable( )

Synopsis

```
INT16 hr_irq_disable(
    HR_IRQ_SRC      irq)                        /* In: The interrupt source */
```

Description

This function is only intended to be used when writing device drivers or other HW-related code.

Disable a specific interrupt.
See function hr_irq_connect for a list of supported IRQ sources.

Returns OK if the irq is disabled.
Returns ERROR if:
- The irq argument is not valid.
- The irq is not connected.

---

hr_irq_enable( )

Synopsis

```
INT16 hr_irq_enable(
    HR_IRQ_SRC      irq)                        /* In: The interrupt source */
```

Description

This function is only intended to be used when writing device drivers or other HW-related code.

Enable a specific interrupt.
See function hr_irq_connect for a list of supported IRQ sources.

Returns OK if the irq is enabled.
Returns ERROR if:
- The irq argument is not valid.
- The irq is not connected.

---

hr_led_appl( )

Synopsis

```
INT16 hr_led_appl(
    INT16*  p_status)                   /* Out: result status - may be NULL */
```

Description

For EOS, all LEDs are controlled entirely by the HW. It is not possible to control any LEDs from SW.
Returns ERROR (always) on EOS devices.

This function directs the control of the SW LED (LED_OK) of the VCU-C2 to the application and prevents CSS from controlling the LED on the device. The control can be directed back to CSS by calling the API hr_led_css().

The LED is turned off before the control is handed over to the application.

p_status may be NULL if detailed information is not required.
Return the status value OK if the function is successful or ERROR if the LED can not be turned off.

Returns OK on success or ERROR if the LEDs can not be turned off.

hr_led_comm_act_set( )

Synopsis

```
INT16 hr_led_comm_act_set(
    UINT16 settings,                                /* In: Bit mask of LED_xx */
    UINT16 led_mask )                               /* In: Bit mask of LED_xx */
```

Description

For EOS, all LEDs are controlled entirely by the HW and it is not possible to control any LEDs from SW.
This function will return ERROR on EOS devices.

This function may be used to assign one or more LEDs to be used as a CSS communication activity LED or to be controlled by the application.

The following LEDs can be switched between CSS communication activity or application control.

Define	Meaning
LED_SERA	SerA LED
LED_SERB	SerB LED
LED_MVB2	MVB2 LED
LED_ETH2	ETH2 LED
LED_MON	MON LED

If bit is set in led_mask then setting bit determines if the led is used for comm activity or not. A bit set in the led_mask parameter indicates which LED to be affected.
A corresponding bit set in the settings parameter indicates if the LED should be used as a CSS communication activity LED or not.

Example on how to turn OFF CSS communication activity for LED SerA:

```
hr_led_comm_act_set ( 0, LED_SERA, &)
```

Example on how to turn ON CSS communication activity for LED SerA:

```
hr_led_comm_act_set ( LED_SERA, LED_SERA, &)
```

Returns OK if the operation was successful.
Returns ERROR if:
- A bit in settings is set without the corresponding bit in led_mask being set.
- A bit is set in either argument that does not correspond to any known LED. - The device does not support a specific LED.
- The LED could not be registered for communication activity.

hr_led_css( )

Synopsis

```
INT16 hr_led_css(
    INT16*  p_status)                    /* Out: result status - may be NULL */
```

Description

For EOS, all LEDs are controlled entirely by the HW and it is not possible to control any LEDs from SW.
This function will return ERROR on EOS devices.

This function directs the control of the SW LED of the VCU-C2 to the system (CSS).
The actual system status of the LED is maintained during any absence of control and will be put out on the LED.

Return the status value OK.
Returns OK.

hr_led_set( )

Synopsis

```
INT16 hr_led_set(
    UINT16  pattern,            /* In:  led pattern to be set on or off */
    UINT16  mask,               /* In:  bit mask used to select which LED(s)
                                        shall be controlled */
    INT16  *p_status)           /* Out: result status - may be NULL */
```

Description

For EOS, all LEDs are controlled entirely by the HW and it is not possible to control any LEDs from SW.
This function will return ERROR on EOS devices.

This function sets the LEDs on the device.
It is always possible to modify the application defined LED L1 on the device.
To modify the SW LED the application needs to call the hr_led_appl function first.
To modify the communication activity LEDs the application needs to call the hr_led_comm_act_set first.

The parameter pattern is a bitset that defines the LED pattern to set.

The parameter mask is a bitset that specifies a subset of all LEDs which state to change.

Possible value for pattern and mask:

VCU-C2 LED	Define	Value	Application controllable
ERR	LED_ER	0x0002	No
MVB1	LED_MVB	0x0008	No
SW	LED_OK	0x0010	Yes (use hr_led_appl first)
ETH1	LED_ETH	0x0020	No
USB1	LED_USB1	0x0040	No
L1	LED_L1	0x0100	Yes
SerA	LED_SERA	0x0200	Yes (use hr_led_comm_act_set first)
SerB	LED_SERB	0x0400	Yes (use hr_led_comm_act_set first)
MVB2	LED_MVB2	0x0800	Yes (use hr_led_comm_act_set first)
ETH2	LED_ETH2	0x1000	Yes (use hr_led_comm_act_set first)
USB2	LED_USB2	0x2000	No
MON	LED_MON	0x4000	Yes (use hr_led_comm_act_set first)

Bitwise OR operation is used to specify more that one LED.

Example on how to turn on LED_OK and turn off LED_L1:

```
hr_led_set ( LED_OK, LED_OK | LED_L1, &status )
```

Example on how to turn on LED_L1 and LED_OK but turn off LED_SERA and LED_SERB:

```
hr_led_set ( LED_L1 | LED_OK,
             LED_L1 | LED_OK | LED_SERA | LED_SERB,
               &status )
```

The state of the other LEDs are not changed.

Return the following status and return value:

Return value	p_status	Description
ERROR	ERROR	hr_led_appl() or hr_led_comm_act_set
-	-	has not been called
-	-	or the LED operation failed.
OK	OK	The LED operation was successful.

hr_line_trip_is_enabled( )

Synopsis

```
STATUS hr_line_trip_is_enabled(
        BOOL  *p_enabled)     /* Out: State of Line Trip relay */
                              /*      TRUE: Line trip relay enabled(active) */
                              /*      FALSE: Line trip relay disabled(inactive) */
```

Description

This function reads the status of the line trip relay.

Returns OK if the operation was successful, otherwise ERROR.

Note: This function is not supported in VCU-C2 and will return ERROR if used.

hr_line_trip_set_disable( )

Synopsis

```
STATUS hr_line_trip_set_disable(
        void)
```

Description

This function disables(de-activates) the Line trip relay.

Returns OK if the operation was successful, otherwise ERROR.

Note: This function is not supported in VCU-C2 and will return ERROR if used.

hr_line_trip_set_enable( )

Synopsis

```
STATUS hr_line_trip_set_enable(
    void)
```

Description

This function enables(activates) the Line trip relay.

Returns OK if the operation was successful, otherwise ERROR.

Note: This function is not supported in VCU-C2 and will return ERROR if used.

hr_memory_barrier( )

Synopsis

```
void hr_memory_barrier(
    void)
```

Description

This function is only intended to be used when writing device drivers or other HW-related code.

This function ensures that all already started memory I/O operations have completed before returning.

hr_mvb_service_set( )

Synopsis

```
INT16 hr_mvb_service_set(
    BOOL   activate,            /* In:  TRUE:  switch on re-direction
                                        FALSE: switch off re-direction */
    INT16  *p_status)           /* Out: result status */
```

Description

This function does nothing.
Returns the following value and status code:

Return value	*p_status	Description
ERROR	NOT_SUPPORTED	Not supported on the device

hr_region_valid( )

Synopsis

```
INT16 hr_region_valid(
    char  *start,        /* In: Start of region */
    char  *end,          /* In: End of region */
    BOOL  readonly)      /* In: Indicates if region is read-only or not */
```

Description

Check if a memory region is valid.

If the read-only flag is TRUE this function returns OK if the specified memory region [start, end] is readable, else ERROR.

If the read-only flag is FALSE this function returns OK if the specified memory region is readable and writable, else ERROR.

Function is not supported in CSS4 and only kept for backward compatibility.

Returns ERROR (always).

hr_reset_reason_get( )

Synopsis

```
INT16 hr_reset_reason_get(
    UINT16  *p_reset_reason,     /* Out: reset reason */
    INT16   *p_status)           /* Out: result status */
```

Description

This function is used after a reset to determine the reason of the reset.

*p_reset_reason can have the following value:

Reset reason	Value	Description
HR_RESET_REASON_MODE_CHANGE	0	Power on or reset to IDLE or RUN
HR_RESET_REASON_ERROR	1	Software or exception reset
HR_RESET_REASON_SW_WATCHDOG	2	Watchdog reset
HR_RESET_REASON_ECC_ERROR	3	Uncorrectable ECC error

Returns OK and *p_status is set to OK always.

hr_stall_set( )

Synopsis

```
INT16 hr_stall_set(
    BOOL   activate,            /* In:  TRUE:  activate stall alarm
                                        FALSE: deactivate stall alarm */
    INT16  *p_status)           /* Out: result status */
```

Description

This function activates or deactivates the stall alarm relay on the device.

If the parameter activate is TRUE the stall alarm relay is activated, else it is deactivated.

When using this function the following should be taken into consideration:

- This function must be called repeatedly at least every 300 ms in order to keep the stall relay activated.
- Due to long execution time (approximate 50 ms), this function should not be be called from any vital task with short cycle time or from interrupt context.
- There is no feedback from the relay contacts that they are actually according to the requested state, thus the stall relay should not be used in any safety related function.

Returns the following value and status code:

Return value	*p_status	Description
ERROR	ERROR	If the I2C access fails
OK	OK	On success

Note: This function is not supported in CSS4 for EOS and if used it will always return the following value and status code:

Return value	*p_status	Description
OK	NOT_SUPPORTED	Not supported on the device

hr_temp_100_lower_limit_get( )

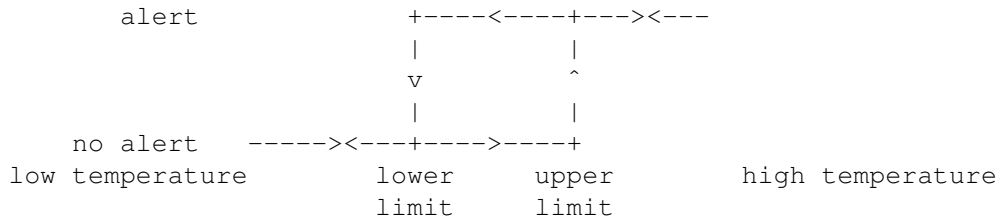
Synopsis

```
STATUS hr_temp_100_lower_limit_get(
    UINT16  sensor_id,      /* In: The id of the requested temperatur sensor */
    INT16   *limit)         /* Out: 0.01 degree celsius C */
```

Description

Get the lower temperature alert limit.

On EOS, the temperature alert behaviour can be described as a hysteresis. When the temperature value is higher than the upper limit, the alert will be active. When the temperature value is lower than the lower limit, the alert will be inactive.



Note: This function is not supported in VCU-C2 and will return ERROR if used.

hr_temp_100_lower_limit_set( )

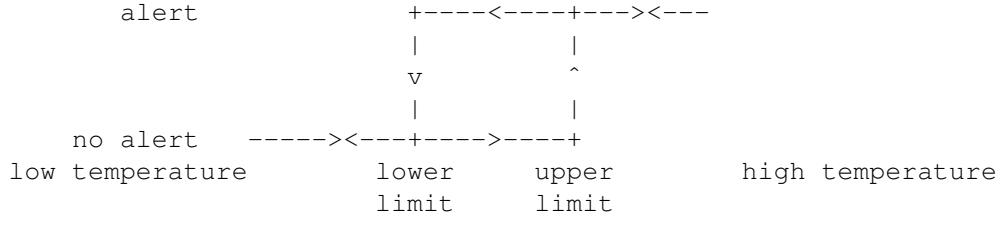
Synopsis

```
STATUS hr_temp_100_lower_limit_set(
    UINT16  sensor_id,      /* In: The id of the requested temperatur sensor */
    INT16   limit)          /* In: 0.01 degree celsius C */
```

Description

Set the lower temperature alert limit.

On EOS, the temperature alert behaviour can be described as a hysteresis. When the temperature value is higher than the upper limit, the alert will be active. When the temperature value is lower than the lower limit, the alert will be inactive.



Note: This function is not supported in VCU-C2 and will return ERROR if used.

hr_temp_100_status_get( )

Synopsis

```
INT16 hr_temp_100_status_get(
    UINT16  sensor_id,              /* In:  The id of the temperature sensor */
    INT16   *p_temp,                /* Out: 1/100 dgr. C */
    INT16   *p_status)              /* Out: result status */
```

Description

Read the requested sensors temperature and status.
The temperature is scaled by 100, i.e. the resolution is 0.01 dgr. C. Note: The physical temp sensor may have lower resolution.

The following values on the parameter sensor_id are valid:

TEMP_SENSOR_VCUC_CPU_1	VCU-C2 CPU board temperature sensor 1
TEMP_SENSOR_VCUC_CPU_2	VCU-C2 CPU board temperature sensor 2
HR_TEMPERATURE_SENSOR_1	EOS CPU temperature sensor
HR_TEMPERATURE_SENSOR_2	EOS digital out temperature sensor
HR_TEMPERATURE_SENSOR_3	EOS fiber optics temperature sensor

Returns the following value and status code:

Return value	***p_status**	**Description**
OK	OK	Temperature OK.
ERROR	HR_TEMP_BAD	Too hot! more than 85 dgr. C on VCU-C2
-	-	On EOS: limit can be changed by application
ERROR	ERROR	The temperature can not be read or the sensor_id is invalid or
-	-	p_temp is NULL.

hr_temp_100_upper_limit_get( )

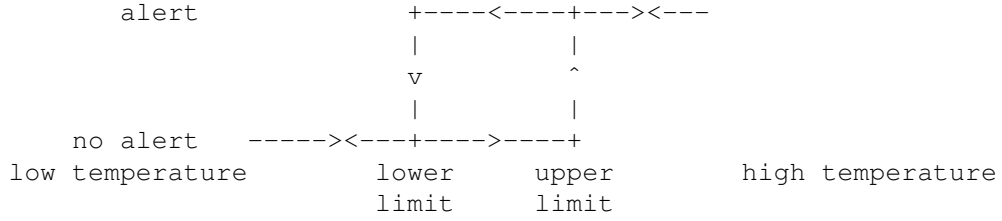
Synopsis

```
STATUS hr_temp_100_upper_limit_get(
    UINT16  sensor_id,      /* In: The id of the requested temperatur sensor */
    INT16   *limit)         /* Out: 0.01 degree celsius C */
```

Description

Get the upper temperature alert limit.

On EOS, the temperature alert behaviour can be described as a hysteresis. When the temperature value is higher than the upper limit, the alert will be active. When the temperature value is lower than the lower limit, the alert will be inactive.



Note: This function is not supported in VCU-C2 and will return ERROR if used.

hr_temp_100_upper_limit_set( )

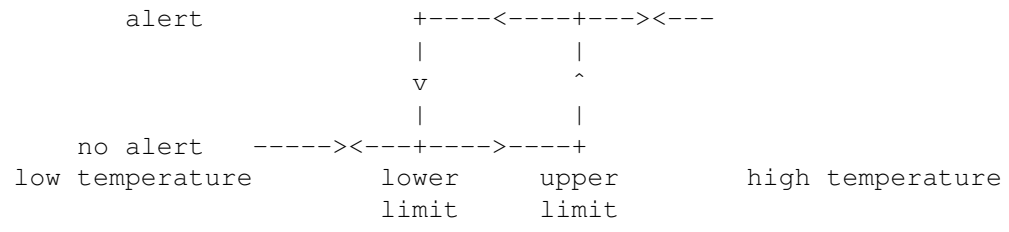
Synopsis

```
STATUS hr_temp_100_upper_limit_set(
    UINT16  sensor_id,      /* In: The id of the requested temperatur sensor */
    INT16   limit)          /* In: 0.01 degree celsius C */
```

Description

Set the upper temperature alert limit.

On EOS, the temperature alert behaviour can be described as a hysteresis. When the temperature value is higher than the upper limit, the alert will be active. When the temperature value is lower than the lower limit, the alert will be inactive.



Note: This function is not supported in VCU-C2 and will return ERROR if used.

---

hr_temp_status_get( )

Synopsis

```
INT16 hr_temp_status_get(
    INT16  *p_temp,                    /* Out: dgr. C */
    INT16  *p_status)                  /* Out: result status - may be NULL */
```

Description

Read temperature and status on the device.
The temperature resolution is 1 dgr. C.

Returns the following value and status code:

Return value	*p_status	Description
OK	OK	Temperature OK
ERROR	HR_TEMP_BAD	Too hot! more than 85 dgr. C on VCU-C2
-	-	On EOS: limit can be changed by application
ERROR	ERROR	The temperature can not be read or p_temp is NULL.

---

hr_temp_status_param_get( )

Synopsis

```
INT16 hr_temp_status_param_get(
    UINT16  sensor_id,             /* In:  The id of the temperature sensor */
    INT16   *p_temp,               /* Out: dgr. C */
    INT16   *p_status)             /* Out: result status - may be NULL */
```

Description

Read the requested sensors temperature and status.
The temperature resolution is 1 dgr. C.

The following values on the parameter sensor_id are valid:

TEMP_SENSOR_VCUC_CPU_1	VCU-C2 CPU board temperature sensor 1
TEMP_SENSOR_VCUC_CPU_2	VCU-C2 CPU board temperature sensor 2
HR_TEMPERATURE_SENSOR_1	EOS CPU temperature sensor
HR_TEMPERATURE_SENSOR_2	EOS digital out temperature sensor
HR_TEMPERATURE_SENSOR_3	EOS fiber optics temperature sensor

Returns the following value and status code:

Return value	*p_status	Description
OK	OK	Temperature OK
ERROR	HR_TEMP_BAD	Too hot! more than 85 dgr. C on VCU-C2
-	-	On EOS: limit can be changed by application
ERROR	ERROR	The temperature can not be read or the sensor_id is invalid or
-	-	p_temp is NULL.

hr_test_hw_reset_status_get( )

Synopsis

```
INT16 hr_test_hw_reset_status_get(
    INT16   *pStatus)           /* Out: Result of 'Test of HW Reset' */
```

Description

Get the result of the last 'Test of HW Reset'

Returns the following status codes:

Return value	*pStatus	Description
OK	OK	'Test of HW Reset' is working
OK	ERROR	'Test of HW Reset' is not working
ERROR	N/A	Result of 'Test of HW Reset' is not available

hr_timestamp( )

Synopsis

```
INT16 hr_timestamp(
    UINT32  *p_result)           /* Out: The current timestamp timer tick count. */
```

Description

Return the device specific timestamp timer tick in microsecond.

Returns OK on success or ERROR if the timestamp timer is not enabled.

hr_timestamp_disable( )

Synopsis

```
INT16 hr_timestamp_disable(
    void)
```

Description

Disables the device specific timestamp timer.

Returns OK on success or ERROR if the timestamp timer is not enabled.

hr_timestamp_enable( )

Synopsis

```
INT16 hr_timestamp_enable(
    void)
```

Description

Enables and resets the device specific timestamp timer. If this function is called consecutive times, without calling hr_timestamp_disable in between, the timestamp functionality will only be initialized the first time and the rest of the calls the timer will be reset only.

Returns OK on success or ERROR if the timestamp timer is not available.

hr_traffic_mem_addr_size_get( )

Synopsis

```
INT16 hr_traffic_mem_addr_size_get(
    void           **pp_addr,   /* Out: traffic memory base address */
    unsigned long  *p_size,     /* Out: traffic memory size */
    INT16          *p_status)   /* Out: result status */
```

Description

Get the base address and size for the traffic store memory for MVB1.

Returns OK and *p_status is set to OK always.

---

hr_wdog_appl( )

Synopsis

```
INT16 hr_wdog_appl(
    INT16  *p_status)    /* Out: result status */
```

Description

This functionality lets the application take control over the watchdog trigging.

The application uses hr_wdog_trigger to trig the watchdog until the control is handed over to CSS.

The first trigging of the watchdog is made by this function.

Returns the following value and status code:

Return value	*p_status	Description
OK	OK	Application has now control over the watchdog

---

hr_wdog_css( )

Synopsis

```
INT16 hr_wdog_css(
    INT16  *p_status)    /* Out: result status */
```

Description

This functionality lets CSS take control over the watchdog trigging again.

Returns the following value and status code:

Return value	*p_status	Description
ERROR	ERROR	Watchdog task can not be activated
OK	OK	CSS has taken control over the watchdog

---

hr_wdog_trigger( )

Synopsis

```
INT16 hr_wdog_trigger(
    INT16  *p_status)    /* Out: result status */
```

Description

With this routine, the application triggers the hardware watchdog.

Returns the following value and status code:

Return value	*p_status	Description
OK	OK	Application has trigged the watchdog
ERROR	ERROR	The application has no control over the watchdog

2.13 IP - Internet Protocol Service

The CSS IP service is publishing a selection of API funtions from the VxWorks network related Libraries.
The following functions are available, for description refer to:

VxWorks Kernel API Reference, 6.9
VxWorks Application API Reference, 6.9
Wind River Network Stack Kernel API Reference, 6.9
Wind River Network Stack Application API Reference, 6.9

sockLib	ioLib	inetLib	inetAddrLib	hostLib	selectLib
accept	close	inet_ntoa_b	inet_addr	hostGetByName	select
bind	ioctl				
connect	open				
connectWithTimeout	read				
getpeername	write				
getsockname					
getsockopt					
listen					
recv					
recvfrom					
recvmsg					
send					
sendmsg					
sendto					
setsockopt					
shutdown					
socket					

In addition to the standard functions the IP service also provides the following functions:

ip_status_get	Poll the status of a network interface. Optionally wait until the network becomes ready to use.
ip_network_info	Get the IP address of a network interface.
ip_eth_addr_get	Get the MAC address of a network interface.
ip_vlan_create	Create a virtual VLAN interface
ip_vlan_delete	Delete a virtual VLAN interface

These functions are described in the MITRAC CC CSS API Reference Manual.

VLAN support:

CSS supports IP communication using tagged VLAN Ethernet frames.

Note that the IP network infrastructure must also support VLANs.
Note that the max MTU is decreased by 4 in order to accommodate the VLAN header in the Ethernet frame.

There are two ways to transmit VLAN frames:

Using setsockopt to convert an already opened socket into a socket-based VLAN.
Opening a socket in the address range defined by the IP-address and netmask of a VLAN interface.

A VLAN interface is needed in order to receive VLAN frames. VLAN frames sent to a device with no VLAN interface are simply dropped.

A VLAN interface is created and deleted by the following two API functions:

ip_vlan_create	Create a VLAN interface based on the arguments.
ip_vlan_delete	Delete a VLAN interface.

Overlapping VLAN subnets:

Normally a VLAN interface will be on a different subnet than the physical interface, but it is perfectly possible to use the same IP-address and netmask on the VLAN interface as on the real interface.
Received VLAN frames are always routed through the VLAN interface.
In order to send VLAN frames in such a scenario the application must use setsockopts() to turn the socket into a VLAN socket. This can be done as in the following example:

```
int             sock      = socket( AF_INET, SOCK_DGRAM, 0 );
struct sovlan   vlan_opts = {  1,          /* Enable VLAN */
                               FALSE,      /* Disable priority tags */
                               123,        /* The VLAN ID */
                               0 };        /* Priority not used */

setsockopt( s, SOL_SOCKET, SO_VLAN, (char*)&vlan_opts, sizeof(vlan_opts) );
```

Definitions used by the published API:

Definitions related to Sockets	Value	Description
SOCK_STREAM	1	Stream socket
SOCK_DGRAM	2	Datagram socket
SOCK_RAW	3	Raw-protocol interface
MSG_OOB	0x1	Process out-of-band data
MSG_PEEK	0x2	Peek at incoming message
MSG_DONTROUTE	0x4	Send without using routing tables
SOL_SOCKET	0xffff	Options for socket level
Address families		
AF_INET	2	Internetwork: UDP, TCP, etc.
POSIX Error codes		
ENOTSUP	35	Unsupported value
EISCONN	56	Socket is already connected
EINPROGRESS	68	Operation now in progress
EALREADY	69	Operation already in progress
Option flags per-socket		
SO_DEBUG	0x0001	Turn on debugging info recording
SO_REUSEADDR	0x0004	Allow local address reuse
SO_KEEPALIVE	0x0008	Keep connections alive
SO_BROADCAST	0x0020	Permit sending of broadcast msgs
SO_LINGER	0x0080	Linger on close if data present
SO_REUSEPORT	0x0200	Allow local address and port reuse
SO_VLAN	0x8000	Get/set VLAN socket options
Option values per-socket		
TCP_LINGERTIME	120	Default linger time (s) when enabled
Additional options		
SO_SNDBUF	0x1001	Send buffer size
SO_RCVBUF	0x1002	Receive buffer size
SO_BINDTODEVICE	0x1010	Bind to specific device
SO_OOBINLINE	0x1011	Leave received OOB data in line
User-settable options		
TCP_NODELAY	0x01	Do not delay send to coalesce packets
TCP_MAXSEG	0x02	Set maximum segment size

Options for use with [gs]etsockopt		
IP_OPTIONS	1	Set/get IP options
IP_HDRINCL	2	Header is included with data
IP_TOS	3	IP type of service and preced
IP_TTL	4	IP time to live
IP_RECVRETOPTS	6	Receive IP opts for response
IP_RECVDSTADDR	7	Receive IP dst addr w/dgram
IP_MULTICAST_IF	9	Set/get IP multicast i/f
IP_MULTICAST_TTL	10	Set/get IP multicast ttl
IP_MULTICAST_LOOP	11	Set/get IP multicast loopback
IP_ADD_MEMBERSHIP	12	Add an IP group membership
IP_DROP_MEMBERSHIP	13	Drop an IP group membership

Definitions for IP type of service	
IPTOS_LOWDELAY	0x10
IPTOS_THROUGHPUT	0x08
IPTOS_RELIABILITY	0x04
IPTOS_MINCOST	0x02

Cluster defines	
CL_SIZE_128	128
MLEN	CL_SIZE_128

Protocols		
IPPROTO_IP	0	Dummy for IP
IPPROTO_TCP	6	TCP

VxWorks definitions used by the published API:

Definition	**Value**	**Description**
INET_ADDR_LEN	18	Length of ASCII represention
NBBY	8	Number of bits in a byte
FD_SETSIZE	2048	The number of sockets in the system

Arguments to shutdown():

Definition	**Value**	**Description**
SHUT_RD	0	Shut down the reading side
SHUT_WR	1	Shut down the writing side
SHUT_RDWR	2	Shut down both sides

Old vxWorks macro definitions used by the published API:

Macro	**Description**
NFDBITS	Bits per mask
howmany(x, y)	
FD_SET(n, p)	
FD_CLR(n, p)	
FD_ISSET(n, p)	
FD_ZERO(p)	

Module number and status messages used by the published API:

Module number related to	**Value**
selectLib	
M_selectLib	(57 << 16)
S_selectLib_NO_SELECT_SUPPORT_IN_DRIVER	(M_selectLib | 1)
S_selectLib_NO_SELECT_CONTEXT	(M_selectLib | 2)
S_selectLib_WIDTH_OUT_OF_RANGE	(M_selectLib | 3)
hostLib	
M_hostLib	(50 << 16)
S_hostLib_UNKNOWN_HOST	(M_hostLib | 1)
S_hostLib_INVALID_PARAMETER	(M_hostLib | 3)

Macros for number representation conversion:

ntohl(x)
ntohs(x)
htonl(x)
htons(x)

NTOHL(x)
NTOHS(x)
HTONL(x)
HTONS(x)

Type definitions used by the published API:

Type reference	**Own type name**
vxWorks types	
typedef char*	caddr_t
typedef socklen_t	u_int
typedef unsigned int	in_addr_t;
vxWorks old types	
typedef unsigned char	u_char
typedef unsigned int	u_int
typedef unsigned short	u_short
typedef unsigned long	u_long
typedef long	fd_mask
Structure used by kernel to store most addresses	
typedef SOCKADDR	IP_SOCKADDR
Standard types	
typedef struct timeval	IP_TIMEVAL
typedef struct fd_set	IP_FDSET
typedef struct msghdr	IP_MSGHDR
typedef struct sockaddr	SOCKADDR

Structures used by the published API:

Type	Description
times	
struct timeval {	
long tv_sec;	seconds
long tv_usec;	microseconds
}	
vxWorks old type	
typedef struct fd_set {	
fd_mask fds_bits[howmany(FD_SETSIZE, NFDBITS)];	
}	
uio included in ioLib	
struct iovec {	
caddr_t iov_base;	
size_t iov_len;	
}	
socket	
struct linger {	Structure used for manipulating linger option
int l_onoff;	Option on/off
int l_linger;	Linger time
}	
struct sovlan {	Structure used for manipulating VLAN options
int vlan_onoff;	Option on/off
BOOL priority_tagged;	Priority option enable
unsigned short vid;	The VLAN ID: 1..4094
unsigned short upriority;	The prio: 0..7
}	

```
struct sockaddr {                       Structure used by kernel to store most addresses
        u_char sa_len;                  Total length
        u_char sa_family;               Address family
        char sa_data[14];               Actually longer; address value
}
struct msghdr {                         Message header for recvmsg and sendmsg calls
        void *msg_name;                 Optional address
        socklen_t msg_namelen;          Size of address
        struct iovec *msg_iov;          Scatter/gather array
        int msg_iovlen;                 Number of elements in msg_iov
        void *msg_control;              Ancillary data
        socklen_t msg_controllen;       Ancillary data buffer len
        int msg_flags;                  Flags on received message
}
```

inet

```
struct in_addr {                        Internet address
        unsigned int s_addr;
}
struct sockaddr_in {                    Socket address, internet style
        u_char sin_len;
        u_char sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
}
Argument structure for IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP
struct ip_mreq {
        struct in_addr imr_multiaddr;   IP multicast address of group
        struct in_addr imr_interface;   Local IP address of interface
}
```

---

ip_address_add( )

Synopsis

```
int ip_address_add(
    const char   *if_name,                  /* In: Interface name, e.g. */
                                            /*     motetsec0 or motetsec1 */
    UINT32       ip_address,                /* In: IP address */
    UINT32       net_mask)                  /* In: Net mask */
```

Description

Adds an IP address to a network interface.
The network interface must be initialized.
It is possible to add two additional IP addresses per interface.

Returns OK if the IP address is successfully added.
Returns ERROR if:
- Function argument is NULL.
- The address is not possible to add to the interface.
- The network interface does not exist.
- The network interface is not initialized.
- More than two addresses are added.
- The address does not match the subnet.
- The netmask is invalid.

---

ip_def_gw_addr_get( )

Synopsis

```
int ip_def_gw_addr_get(
    const char *if_name,                    /* In:  The interface name */
    char def_gw_address[])                  /* Out: The default GW address */
```

Description

This function copies the default gateway IP address for an interface specified by if_name into the def_gw_address.

Caller must supply a buffer for the default GW address. The buffer must be large enough to contain the resulting string (Recomended: at least 20 characters).

Returns OK if an gateway address address is found, othewise ERROR.

---

ip_eth_addr_get( )

Synopsis

```
STATUS ip_eth_addr_get(
    const char* if_name,/* In:  Network interface or NULL for primary interface */
    unsigned    mac_addr_len,/* In:  Number of bytes in p_mac_addr */
    UINT8      *p_mac_addr) /* Out: Ethernet address of interface (MAC) */
```

Description

This function is used to poll the MAC address of a specific network interface specified in the parameter if_name.
If the parameter is NULL the primary network interface is used.

The MAC address is copied to the output buffer p_mac_addr, which should be at least 6 bytes long.

Returns OK on success.
Returns ERROR if no interface could be found or any parameter in the call is invalid.

---

ip_if_idx_to_name( )

Synopsis

```
int ip_if_idx_to_name(
        UINT16 idx,     /* In:  Interface index */
        char *name)     /* Out: Interface name */
```

Description

Get the name of the network interface at index.
If the interface exists and is configured then the IF name is returned.

Application must supply a buffer with enough space for the interface_name. Ideally END_NAME_MAX + 1 bytes.

Returns OK if a network interface exist and the name is returned.
Returns ERROR if:
- Function arguments are NULL.
- The interface does not exist or is not available.
- An error occurs when writing the string.

---

ip_is_link_up( )

Synopsis

```
BOOL ip_is_link_up(
    const char *pDevice,  /* In: Device name (i.e. ppp or motetsec). */
    int        unit)      /* In: Unit number  */
```

Description

Gets the current link status for the specified interface.

Returns TRUE if link is up.
Returns FALSE if link is down.

ip_network_info( )

Synopsis

```
int ip_network_info(
    int                idx,                    /* In: Idx into if table */
    char*              interface_name,         /* Out: The if name */
    char*              inet_address,           /* Out: IP adrs */
    IP_NETWORK_INFO_T* p_status)               /* Out: Detailed status */
```

Description

Get information about a network interface.
If the interface exists and is configured then the IP address is returned.
If an interface exist it tries to get the associated IP address.
The value of p_status reflects the result of this function:

IP_NETWORK_INFO_ERROR	interface_name or inet_address is NULL
IP_NETWORK_NOT_FOUND	No interface at index (idx)
IP_NETWORK_DISABLED	Interface is disabled
IP_NETWORK_ENABLED	IP address is available
IP_NETWORK_NO_INFO	IP address is not available

Application must supply a buffer of at least 10 bytes for the interface_name and a buffer of at least 20 bytes for the inet_address information.

Returns OK if a network interface exist and information is returned.
Returns ERROR if:
- Function arguments are NULL.
- The interface does not exist.

ip_status_get( )

Synopsis

```
IP_STATUS_T ip_status_get(
    const char* if_name,                   /* In: The interface name */
    int         timeout)                   /* In: The maximum waiting time */
```

Description

This function is used to poll the status of the network interface specified in the parameter if_name.
If the parameter is NULL the primary network interface is used.

In the parameter timeout it can be specified how long to wait for the status in ms. The function will wait until a connection is established on the network interface before the status is returned. The function will however not wait more than timeout ms.
If timeout is OS_WAIT_FOREVER the call will block until a connection is established.
If timeout is OS_NO_WAIT the call will return immediately with the status of the link.

The function returns the following status:

IP_ERROR	if the network interface is not supported
IP_ERROR	if the network interface could not be found
IP_ERROR	if the network interface is not initialized
IP_ERROR	if called from interrupt context
IP_NO_STATUS	if no network connection is established during the timeout
IP_NO_NETWORK	if no network configuration is specified for that interface
IP_LOOPBACK	if the device is configured in loopback mode, i.e the transmit and receive signals are connected in the driver.
IP_RUNNING	if a network link for the interface is established

ip_vlan_create( )

Synopsis

```
int ip_vlan_create(
    const char *vlan_name,                  /* In: Name of VLAN interface */
    UINT32      if_idx,                     /* In: 0 => ETH1, 1 => ETH2 */
    UINT16      vlan_id,                    /* In: The VLAN ID */
    const char *ip_addr,                    /* In: The IP address */
    const char *ip_mask )                   /* In: The IP netmask */
```

Description

This function creates a VLAN interface and connects it to a real Ethernet interface.

Returns OK if the interface could be created.
Returns ERROR if:
- vlan_name is not unique.
- if_idx is out of range.
- if_idx == 1, but device has only one Ethernet interface.
- Invalid vlan_id.
- ip_addr or ip_mask (or both) is NULL.
- A virtual device could not be created.

---

ip_vlan_delete( )

Synopsis

```
int ip_vlan_delete(
    const char *vlan_name )                 /* In: The VLAN interface name */
```

Description

This function deletes the virtual interface previously created by ip_vlan_create.

Returns OK if the device was successfuly removed.
Returns ERROR if:
- Argument vlan_name is NULL.
- A virtual device with this name does not exist.
- The virtual interface could not be removed.

2.14 LOAD - CPU Load Measurement Service

The load service in CSS periodically computes the total CPU load and the load of each task and the interrupts.

The CPU load is defined as: ”the load of all tasks, except rts_tcn_bg”.

The results are presented on 6 signals:

CSS_LOAD::CPULoadCurrent
CSS_LOAD::CPULoadAverage
CSS_LOAD::IntLoadCurrent
CSS_LOAD::IntLoadAverage
CSS_LOAD::TaskLoadCurrent
CSS_LOAD::TaskLoadAverage

These signals are updated by the load_task, normaly every second, but that may be changed by calling function load_sample_rate_set. The TaskLoadCurrent and TaskLoadAverage signals are only updated if a task has been specified using the function load_task_add.

The load measurement is very precise because it catches every task switch and every interrupt in the system.

---

load_connect_task_signal( )

Synopsis

```
int load_connect_task_signal(
    const char *task_name,                  /* In: The task name */
    FLOAT32    *p_curr_signal,              /* In: The current load signal */
    FLOAT32    *p_aver_signal)              /* In: The average load signal */
```

Description

This routine connects application signals to receive the current and average load of a specific task.
Sending NULL will disconnect the signals.

Returns OK if the signals were properly connected/disconected.
Returns ERROR if task is not known to the load service.

---

load_counter_reset( )

Synopsis

```
void load_counter_reset(
    void)
```

Description

This routine resets the average load counters.

---

load_sample_rate_set( )

Synopsis

```
int load_sample_rate_set(
    UINT32 sample_rate_ms)                  /* In:  The new sample rate */
```

Description

This routine sets the CPU load sampling period in ms.
This period effects the update rate of the signals in the symbol table CSS_LOAD.

Sampling periods between 200 and 120000 ms are allowed, 1000 ms is the default sampling period.

The routine returns ERROR if the samling rate is outside the allowed range otherwise OK.

---

load_task_add( )

Synopsis

```
int load_task_add(
    const char* task_name)                          /* In: The task name */
```

Description

This routine adds the task specified in the parameter task_name to the CPU load measurement service.
If the task does not exist it clears any previously added task.

If the task does not exist routine returns ERROR otherwise OK.

---

load_task_find( )

Synopsis

```
const char* load_task_find(
    void)
```

Description

This routine returns the name of the task that is added to the CPU load measurement service.

If no task is added the routine returns NULL.

---

load_task_remove( )

Synopsis

```
void load_task_remove(
    void)
```

Description

This routine removes the task that is added to the load measurement service.

If no task is currently added the call has no effect.

2.15 MFULW - MCG Framework File Upload Library Wrapper Service

Overview

The MCG Framework File Upload library Wrapper (MFULW) Service provides a C interface for using the MCG Framework File Upload library (McgFwFu) written in C++.

The McgFwFu class, described in [MFUUM], provides the functionality to upload files to the MCG Framework according to [MCGEDSICD]. The basics to use the McgFwFu class is to instantiate a new McgFwFu object with the required parameters and call one of the class uploadFile() methods.

The MFULW Service exposes an API which wrappes the class funcions, including the class instantiation, of the McgFwFu library as described in [MFUUM] with the following exceptions:

- isEdsUriOk() (function not supported by the McgFwFu library on CSS)
- isFileNameOk() (function not supported by the McgFwFu library on CSS).

Simple How To

1. Call the mfulw_getMcgFwFu() function to create a McgFwFu object given all required arguments.
   Remark:
   I you want to use the default logging mechanism of the McgFwFu object pass NULL as the argument for the logger interface parameter. The default logging mechanism will log to the CSS AE-log. If you choose to use your own logging mechanism it must be implemented as a derived class of the LoggerInterface described in [MFUUM].
2. Check the McgFwFu object creation result for any error using the mfulw_getInstanceResult() function.
   Remark:
   If there is an error during the object creation, the object has to be destroyed and a new one has to be created with corrected arguments.
3. Use mfulw_uploadFile() or mfulw_uploadFileUri() to initiate a file upload to the MCG Framework. These functions are blocking and the complete file transfer are handled inside these functions.
   Important Remark:
   These functions has to be called repeatedly until they either returns TRUE (file upload succeeded) or the returned result is equal to RC_UploadFailedMaxAttemptsReached otherwise the internal state of the McgFwFu object is not reset properly. Up to 4 attempts have to be performed according to [MCGEDSICD].
4. If a file upload attempt fails, use the mfulw_sendFileTransferStatus() function to send an appropriate AppResultCode back to the MCG Framework.

The following typedefs are used throughout the use of the MFULW API:

```
typedef enum
{
    Max_Warning                         =   1,
    RC_FileUploaderOk                   =   0,
    RC_InvalidMcgFwUri                  =  -1,
    RC_InvalidLedsUri                   =  -2,
    RC_InvalidMdQueue                   =  -3,
    RC_InvalidLoggerObj                 =  -4,
    RC_InvalidState                     =  -5,
    RC_InvalidFileName                  =  -6,
    RC_FileNotFound                     =  -7,
    RC_UploadOk                         =   0,
    RC_UploadFailed                     =  -8,
    RC_UploadFailedIPTCom               =  -9,
    RC_UploadFailedTimeout              = -10,
    RC_UploadFailedMaxAttemptsReached   = -11,
    RC_InstanceStopped                  = -12,
    RC_Max_Error                        = -13
} ResultCode;
```

Description: Enumeration of result codes returned by the use of the MFULW functions: mfulw_uploadFile(), mfulw_uploadFileUri() and mfulw_sendFileTransferStatus(). Use the function mfulw_getResultCodeString() for getting the corresponding result code description.

```
typedef enum
{
    ARC_NoDataAvailable   =   3,
    ARC_SrvTmpUnavailable =   2,
    ARC_Ok                =   0,
    ARC_IncompVersion     =  -1,
    ARC_SyscallError      =  -4,
    ARC_UnspecError       =  -5,
    ARC_SrvNotSupported   =  -6,
    ARC_NoSubscribers     =  -7,
    ARC_InvalidSrcUri     =  -8,
    ARC_InvalidParam      =  -9,
    ARC_SrvFailed         = -20,
    ARC_ReqFailed         = -21,
    ARC_CRC32Error        = -50,
    ARC_FileExistsError   = -51,
    ARC_NoTransaction     = -52,
    ARC_DiskFull          = -53,
    ARC_FtpError          = -54,
    ARC_InvalidFileSize   = -56
} AppResultCode;
```

Description: Enumeration of application result codes returned by the use of the MFULW functions: mfulw_mfulw_uploadFile(), mfulw_uploadFileUri() and mfulw_sendFileTransferStatus(). Use the function mfulw_getAppResultCodeString() for getting the corresponding application result code description.

```
typedef enum
{
    LI_LL_ERROR     = 0,
    LI_LL_WARNING   = 1,
    LI_LL_INFO      = 2,
    LI_LL_EINFO     = 3,
    LI_LL_DEBUG     = 4,
    LI_LL_INTERNAL1 = 5,
    LI_LL_INTERNAL2 = 6
} LI_LogLevel;
```

Description: Enumeration which defines logging levels. In case the McgFwFu class is instantiating without providing a logger interface these are the possible logging levels that can be set using the mfulw_setLogLevel() function.

---

mfulw_destroyMcgFwFu( )

Synopsis

```
void mfulw_destroyMcgFwFu(
    McgFwFu_t obj)
```

Description

Function for deleting an instance of a McgFwFu object.

Call mfulw_shutdown() function to shutdown any ongoing file upload before the McgFwFu object is deleted using this function.

Parameters:

obj In: Pointer to a McgFwFu object.

Returns none.

---

mfulw_getAppResultCodeString( )

Synopsis

```
const char *mfulw_getAppResultCodeString(
    AppResultCode appResultCode)
```

Description

Returns an application result code description.

Parameters:

appResultCode In: AppResultCode to return back the corresponding description string.

Returns a constant char pointer to a 0-terminated description string for the given aplication result code.

---

mfulw_getInstanceResult( )

Synopsis

```
ResultCode mfulw_getInstanceResult(
    McgFwFu_t obj)
```

Description

Function for getting the result code of a McgFwFu object after it has been created.

It is recomended to call this function after the creation of a McgFwFu object have been made with a call to the mfulw_getMcgFwFu() function. This is because even though the instantiation of the McgFwFu object succeeded it might still be in an internal error state caused by invalid arguments.

Parameters:

obj In: Pointer to a McgFwFu object.

Returns the result code of a McgFwFu object after it has been created.

---

mfulw_getMcgFwFu( )

Synopsis

```
McgFwFu_t mfulw_getMcgFwFu(
    const char *mcgFwUri,
    UINT32 mfUriLen,
    const char *ledsUri,
    UINT32 ledsUriLen,
    MD_QUEUE mdQueueId,
    LoggerIF_t pLogger)
```

Description

Function for creating an instance of a McgFwFu object. This function must be called before any of the following functions are to be used:

- mfulw_destroyMcgFwFu()
- mfulw_getInstanceResult()
- mfulw_uploadFile()
- mfulw_uploadFileUri()
- mfulw_sendFileTransferStatus()

- mfulw_shutdown()
- mfulw_setLogLevel()

After the McgFwFu object have been created it is recomended to call the function mfulw_getInstanceResult() to get the result code of the instantiated McgFwFu object. This is because even though the instantiation of the McgFwFu object succeeded it might still be in an internal error state caused by invalid arguments.

Remark:
An instance of the McgFwFu object has the capability to log various events during a file upload. During the creation of the McgFwFu object it is possible to pass an additional instance of a logger object which will be used by the McgFwFu object. Such a logger instance must be implemented as a derived class of type LoggerInterface described in [MFUUM].
However, if the parameter pLogger is passed as NULL the McgFwFu object will log all events to the AE-log in CSS thus eliminating the need of implementing a logger class.

Parameters:

mcgFwUri	In: Pointer to a 0-terminated string, which contains the URI of the McgFamework where to send files.
mfUriLen	In: The length of the mcgFrameworkUri string - without ending 0-terminating character.
ledsUri	In: Pointer to a 0-terminated string, which contains the full qualified URI of the connected Limited End Device Service (leds) to be used towards the MCG Framework.
ledsUriLen	In: The length of the ledsUri string - without ending 0-terminating character.
mdQueueId	In: ID of the IPTCom MD Queue to be used for receiving messages from the MCG Framework. The listener URI has to be added before calling this function and it has to be the given ledsUri. The MD queue should only be used for the McgFwFu object exclusively.
pLogger	In: Pointer to the logger instance used by the McgFwFu object. If NULL is passed as argument the McgFwFu object will use an internal logger that will use the CSS AE-log.

Returns a pointer of type McgFwFu_t of a McgFwFu object or NULL if the creation of the McgFwFu object failed, e.g. out of memory condition.

mfulw_getResultCodeString( )

Synopsis

```
const char *mfulw_getResultCodeString(
    ResultCode result)
```

Description

Returns a result code description.

Parameters:

result	In: ResultCode to return back the corresponding description string.

Returns a constant char pointer to a 0-terminated description string for the given result code.

mfulw_getVersionBuildInfo( )

Synopsis

```
const char *mfulw_getVersionBuildInfo(
    void)
```

Description

Returns the McgFwFu library's version and build information.

Parameters:

None

Returns a constant char pointer to a 0-terminating string containing the McgFwFu library's versin and build information.

mfulw_sendFileTransferStatus( )

Synopsis

```
UINT32 mfulw_sendFileTransferStatus(
    McgFwFu_t obj,
    const char* mcgFrameworkUri,
    unsigned int mfUriLen,
    unsigned int waysideRef,
    AppResultCode appResultCode,
    ResultCode result)
```

Description

Function for sending a FileTransferStatus request with appResultCode not OK to the MCG Framework, it then waits for the FileTransferStatus response message and returns its appResultCode.

This function should be used to send an error status back to the MCG Framework in case a file upload attempt has failed.

If mcgFrameworkUri is NULL and mfUriLen is 0 then the MCG URI passed as argument to mfulw_getMcgFwFu() when creating the McgFwFu obejct will be used.

Parameters:

obj	In: Pointer to a McgFwFu object.
mcgFrameworkUri	In: Pointer to a 0-terminated string, which contains the URI of the McgFamework, where to send the file transfer status.
mfUriLen	In: The length of the mcgFrameworkUri string - without ending 0-terminating character.
waysideRef	In: WaysideRef to be sent to MCG Framework. 0 if N/A.
appResultCode	In, Out: In: The application result code to report to MCG Framework/Wayside. Out: Application result code of the FileTransferStatus response received from the MCG Framework.
result	Out: This variable will be filled with the result code of the FileTransferStatus Request send attempt. If it fails, the result variable tells the error. Use the function mfulw_getResultCodeString() for getting the corresponding result code description.

Returns TRUE on sucess and FALSE on failure with the result and the appResultCode set accordingly.

mfulw_setLogLevel( )

Synopsis

```
void mfulw_setLogLevel(
    McgFwFu_t       obj,
    LI_LogLevel     logLevel)
```

Description

Sets the log level of the McgFwFu object when the internal logger interface is used, i.e. the McgFwFu object have been created with the parameter pLogger set to NULL in the call to mfulw_getMcgFwFu().

Parameters:

obj	In: Pointer to a McgFwFu object.
logLevel	In: Log level of McgFwFu object.

Returns none.

mfulw_shutdown( )

Synopsis

```
void mfulw_shutdown(
    McgFwFu_t obj)
```

Description

Triggers the shutdown event to an McgFwFu object.

Calling this function puts the McgFwFu instance into STOPPED state. During a file upload the McgFwFu object internally checks the STOP condition in its uploadFile methods at every 200msec. It will immediately terminate an ongoing file uplaod, if the STOP event is signaled.

Remarks:
Once the STOP event is signaled the McgFwFu object cannot be used any more.

Parameters:

obj	In: Pointer to a McgFwFu object.

Returns none.

---

mfulw_uploadFile( )

Synopsis

```
UINT32 mfulw_uploadFile(
    McgFwFu_t obj,
    const char *absFName,
    UINT32 len,
    UINT32 dataId,
    UINT32 waysideRef,
    ResultCode *result,
    AppResultCode *appResultCode)
```

Description

Function for uploading a file to the MCG framework. The file will be uploaded using the MCG URI passed as an argument when the McgFwFu object was created with a call to mfulw_getMcgFwFu().

This method is blocking. It uploads the given file to the MCG Framework according to [MCGEDSICD]. Since a file upload can take several minutes the caller has to create a task for the file upload, if other activities have to run in parallel.

Important Remark:
This function has to be called repeatedly until it either returns TRUE (file upload succeeded) or the returned result is equal to RC_UploadFailedMaxAttemptsReached otherwise the internal state of the McgFwFu object is not reset properly. Up to 4 attempts have to be performed according to [MCGEDSICD].

Parameters:

obj	In: Pointer to a McgFwFu object.
absFName	In: Pointer to a 0-terminated string, which contains the absolute name (including path) of the file to be uploaded to the MCG Framework.
len	In: The length of the absFileName string - without ending 0-termination character.
dataId	In: The dataId to be used in the RequestToSendFile request to the MCG Framework.
waysideRef	In: The waysideRef to be used in the RequestToSendFile request to the MCG Framework.
result	Out: This variable will be filled with the result code of the file upload attempt. If it fails, the result variable tells the error. Use the function mfulw_getResultCodeString() for getting the corresponding result code description.
appResultCode	Out: This variable will be filled with the AppResultCode from the MCG Framework-EDS interface calls. Use the function mfulw_getAppResultCodeString() for getting the corresponding application result code description.

Returns TRUE on sucess and FALSE on failure with the result and the appResultCode set accordingly.

---

mfulw_uploadFileUri( )

Synopsis

```
UINT32 mfulw_uploadFileUri(
    McgFwFu_t obj,
```

```
const char *mcgFwUri,
UINT32 mfUriLen,
const char *absFName,
UINT32 len,
UINT32 dataId,
UINT32 waysideRef,
ResultCode *result,
AppResultCode *appResultCode)
```

Description

Function for uploading a file to the MCG framework.

This method is blocking. It uploads the given file to the MCG Framework according to [MCGEDSICD]. Since a file upload can take several minutes the caller has to create a task for the file upload, if other activities have to run in parallel.

Important Remark:
This function has to be called repeatedly until it either returns TRUE (file upload succeeded) or the returned result is equal to RC_UploadFailedMaxAttemptsReached otherwise the internal state of the McgFwFu object is not reset properly. Up to 4 attempts have to be performed according to [MCGEDSICD].

Parameters:

obj	In: Pointer to a McgFwFu object.
mcgFwUri	In: Pointer to a 0-terminated string, which contains the URI of the McgFamework where to send the file.
mfUriLen	In: The length of the mcgFrameworkUristring - without ending 0-termination character.
absFName	In: Pointer to a 0-terminated string, which contains the absolute name (including path) of the file to be uploaded to the MCG Framework.
len	In: The length of the absFileName string - without ending 0-termination character.
dataId	In: The dataId to be used in the RequestToSendFile request to the MCG Framework
waysideRef	In: The waysideRef to be used in the RequestToSendFile request to the MCG Framework.
result	Out: This variable will be filled with the result code of the file upload attempt. If it fails, the result variable tells the error. Use the function mfulw_getResultCodeString() for getting the corresponding result code description.
appResultCode	Out: This variable will be filled with the appResultCode from the MCG Framework-EDS interface calls. Use the function mfulw_getAppResultCodeString() for getting the corresponding application result code description.

Returns TRUE on sucess and FALSE on failure with the result and the appResultCode set accordingly.

2.16 MON - Monitor Service

The RTS monitor service provides command interpreters that executes CSS and application-specific functions when given commands through a serial line, an MVB or ethernet connection.

An RTS monitor instance can be started on serial, ethernet and MVB interfaces. It is possible to have several RTS monitor instances running at the same time.
A monitor instance executes all its commands in the context of a monitor task. Each RTS monitor instance has an unique environment pointer MON_ENV which shall be used to perform I/O operations on.
All monitor commands in a device are protected by a global semaphore. This guarantees that all commands are serialized (each device can only run one command at a time), regardless of the number of monitor instances.

The monitor output functions (mon_printf, mon_write and mon_broadcast_printf) serve to provide output synchonization of output made by several tasks to the same monitor instance. This synchronization is not provided in the standard ANSI output functions.

Functions mon_write and mon_printf prints to a specific monitor instance while mon_broadcast_printf prints to ALL monitor instances in a device.

Applications may use standard ANSI input functions to read from the monitor input interface.

---

mon_broadcast_printf( )

Synopsis

```
int mon_broadcast_printf(
    const char* fmt,           /* In: String to be printed */
    ...)                       /* In: Arguments to the fmt string */
```

Description

This function performs synchronized output to all current monitor instances.

mon_broadcast_printf performs a mon_printf to all current monitor instances, thus guaranteeing that the output is synchronized to other tasks printing or writing to the same monitor instance.

The string that should be printed is fmt. It is possible to send arguments to the string (...). The number of arguments is variable, but must correspond to the composition of the format string. Allowed data types of the arguments are all basic data types of ANSI-C like char, int, long including their unsigned variants, as well as floats and pointers.

The formatting of the output string is done once. The resulting string is sent to a separate task performing the actual output using a message queue. Actual calls to mon_printf are done from this separate output task. If the queue becomes full the message will be lost.

Output using this function may be lost, but the function is guaranteed to not hang indefinitely.

The function returns the number of characters written or a negative value if an output error occurs.

---

mon_cmd_add( )

Synopsis

```
INT16 mon_cmd_add(
    const char   long_name[],        /* In: The long command name */
    const char   short_name[],       /* In: The short command name */
    MONFUNCPTR   p_mon_func,         /* In: The function to call */
    const char   long_help[],        /* In: The long help-text used by
                                        the help-command */
    const char   short_help[],       /* In: The short help-text used by
                                        the help-command */
    BOOL         password_protected) /* In: Flag set if command requires password */
```

Description

Note: This function is deprecated. Use mon_command_add instead.

---

mon_cmd_del( )

Synopsis

```
INT16 mon_cmd_del(
    const char short_name[])         /* In: Cmd to delete */
```

Description

Note: This function is deprecated. Use mon_command_del instead.

---

mon_command_add( )

Synopsis

```
INT16 mon_command_add(
    const char   long_name[],        /* In: The long command name */
    const char   short_name[],       /* In: The short command name */
    MON_FUNC_PTR p_mon_func,         /* In: The function to call */
    const char   arg_syntax[],       /* In: Argument syntax help text */
    const char   long_help[],        /* In: The long help-text used by
                                        the help-command */
    const char   short_help[],       /* In: The short help-text used by
                                        the help-command */
    BOOL         password_protected) /* In: Flag set if command requires password */
```

Description

This routine adds a command to the monitor.

Both the long_name and short_name for the command command must be unique.

The function that should be called when the command is typed in the monitor is sent to this function as parameter p_mon_func of type MON_FUNC_PTR.

```
  typedef void (*MON_FUNC_PTR)( MON_ENV*   epp,
                                INT16      argc,
                                char*      argv_p[MON_MAX_NR_OF_ARGC] );
```

Arg_syntax is the help text that describes the arguments to p_mon_func. Help text to the added command is sent as long_help and short_help.

If the command should be password protected the password_protected shall be set to TRUE.

The maximum length of long_name is 15 alphabetical characters or underscore.

The maximum length of short_name is 10 alphabetical character or underscore.

Returns OK on success else ERROR if the parameters in the call is not correct, there is no space left in the list of commands or the command name is not unique.

---

mon_command_del( )

Synopsis

```
INT16 mon_command_del(
    const char short_name[])         /* In: Monitor command to delete */
```

Description

This function delets a command in the list of commands with short_name as a basis.

Returns OK on success or ERROR if the command can not be found in the list of commands.

---

mon_flush( )

Synopsis

```
int mon_flush(
    const MON_ENV* const p_mon)        /* In: Monitor instance - May be NULL */
```

Description

This routine writes to the file any unwritten data for a specified output or update stream for which the most recent operation was not input.
For an input stream the behavior is undefined.
If p_mon is NULL, then stdout is used.

Returns OK on success.
Returns ERROR and sets errno to EIO (5) if p_mon is invalid.
Returns ERROR if End Of File is encountered or a write error occurs.

---

mon_get_fd( )

Synopsis

```
INT16 mon_get_fd(
    const MON_ENV* const p_mon,       /* In:  Monitor instance - May be NULL */
    INT32* p_fd)                      /* Out: File Descriptor */
```

Description

This function gets the file descriptor for p_mon and it is returned as p_fd.
If p_mon is NULL then p_fd is set to the file descriptor of STD_OUT.

Returns OK on success or ERROR if p_fd is NULL or p_mon is invalid.

---

mon_get_fp( )

Synopsis

```
INT16 mon_get_fp(
    const MON_ENV* const p_mon,       /* In:  Monitor Instance - May be NULL */
    MON_FILE** p_fp)                  /* Out: File Pointer of p_mon */
```

Description

This function gets the file pointer for p_mon and it is returned as p_fp.
If p_mon is NULL then p_fp is set to stdout.

Returns OK on success or ERROR if p_fp is NULL or p_mon is invalid.

---

mon_printf( )

Synopsis

```
int mon_printf(
    const MON_ENV* const p_mon,        /* In: Monitor instance - May be NULL */
    const char          *fmt,          /* In: String to be printed */
    ...)                               /* In: Arguments to the fmt string */
```

Description

This function writes the string (fmt) to the monitor instance (p_mon).
If p_mon is NULL then this function prints to stdout, without synchronization.
If p_mon is non-NULL then mon_printf guarantees that the whole string is output before other tasks may print (or write) to the same monitor instance.
The string to be printed is given in the argument fmt. It is possible to send arguments to the string (...). The number of arguments is variable, but must correspond to the composition of the format string. Allowed data types of the arguments are all basic data types of ANSI-C like char, int, long including their unsigned variants, as well as floats and pointers.

Returns the number of characters written upon success.
Returns ERROR and sets errno to EIO (5) if p_mon is invalid.
Returns ERROR if End Of File is encountered or a write error occurs.

mon_validate( )

Synopsis

```
INT16 mon_validate(
    const MON_ENV* const p_mon) /* In: Monitor instance */
```

Description

This function may be used by monitor commands to verify that the monitor instance is still valid. A monitor may become invalid if e.g. an ethernet connection is broken.

Returns OK if the monitor is still valid.
Returns ERROR if the monitor is no longer valid.

mon_write( )

Synopsis

```
int mon_write(
    const MON_ENV* const p_mon, /* In: Monitor instance */
    const char* buf,            /* In: Characters to be written */
    const size_t buf_size)      /* In: Size of character array */
```

Description

This function performs synchronized output to a monitor instance. mon_write guarantees that the whole buffer is output before other tasks may write (or printf) to the same monitor instance.
If p_mon is NULL then mon_write outputs to stdout, without task synchronization.

Returns the number of characters written upon success.
Returns ERROR and sets errno to EIO (5) if p_mon is invalid.
Returns ERROR if End Of File is encountered or a write error occurs.

2.17 MT - Memory Test Service

The Memory Test (MT) Service provides run-time testing of memory.

---

mt_free_ram_get( )

Synopsis

```
INT16 mt_free_ram_get(
    UINT32 *p_free_kb )              /* Out: Free memory in kb (1024 bytes) */
```

Description

Report the amount of free memory (RAM) in kile-bytes (1024 bytes).
Note that the amount of free RAM may vary wildly during execution so applications should exercise caution to not use up all free memory.

Returns OK if the free memory could be determined.
Returns ERROR if the free memory could not be determined or p_free_kb is NULL.

2.18 NTP - Network Time Protocol Service

This service provides the NTP server and client for NTP time synchronisation.

---

ntp_deamon_is_alive( )

Synopsis

```
BOOL ntp_deamon_is_alive(
    void)
```

Description

This routine checks if the CSS NTP deamon is running or not.

The routine can be used to check if the NTP demon is running or not before any attempt to stop/start it is performed. If a start/stop attempt fails an SE-Log message is issued and a check before the call can avoid this.

Returns TRUE if the NTP deamon is running otherwise FALSE.

---

ntp_deamon_start( )

Synopsis

```
int ntp_deamon_start(
    void)
```

Description

This routine starts the CSS NTP deamon.

The NTP configuration is read and the deamon task is started.

The routine is used to restart the NTP deamon after it has been stopped by ntp_deamon_stop().

Returns OK upon success or ERROR if the NTP deamon is already running or the NTP deamon can not be started.

---

ntp_deamon_stop( )

Synopsis

```
int ntp_deamon_stop(
    void)
```

Description

This routine stops the CSS NTP deamon.

The NTP drift statistics is written to the drift file and the NTP deamon task is stopped.

The routine can be used to stop the NTP deamon before it can be restarted by ntp_deamon_start().

Returns OK upon success or ERROR if the NTP deamon is not running.

2.19 NVRAM - Non Volatile RAM Handler Service

This service integrates the NVRAM manager library in CSS.
This document includes a brief description of the NVRAM API.
For a detailed description refer to: [NVRAM]

The NVRAM service maintains up to 100 individually named areas.
Please note that the name of an area should not contain any space characters.

The user should be aware that CSS allocates some segments for internal use.
Each persistent Data Recorder also allocates one segment.

---

nvram_check( )

Synopsis

```
int32_t nvram_check(
    void)
```

Description

Check NVRAM device.

This function performs CRC checks on the device and verifies the integrity of the stored data.

Returns:

- NVRAM_OK
  No errors detected
- NVRAM_ERR_INIT
  The NVRAM library is not initialized
- NVRAM_ERR_CHECKSUM
  Checksum error in device

---

nvram_clear( )

Synopsis

```
int32_t nvram_clear(
    nvram_handle handle)                    /* In: handle of a memory segment */
```

Description

Erase the contents of a memory segment.

Sets all bytes in the memory segment specified by handle to 0. A successfully completed clear operation resets the dirty flag of the segment.

Returns:

- NVRAM_OK
  Success
- NVRAM_ERR_INIT
  The NVRAM library is not initialized
- NVRAM_ERR_HANDLE
  The specified handle is invalid

---

nvram_free( )

Synopsis

```
int32_t nvram_free(
    nvram_handle handle)                    /* In: handle of a memory segment */
```

Description

Free an allocated memory segment.

Releases the memory segment associated with the specified handle. The handle becomes invalid and must not be used afterwards.

Returns:

- NVRAM_OK
  Success
- NVRAM_ERR_INIT
  The NVRAM library is not initialized
- NVRAM_ERR_HANDLE
  The specified handle is invalid

---

nvram_get_dirty_flag( )

Synopsis

```
int32_t nvram_get_dirty_flag(
    nvram_handle handle)                    /* In: handle of a memory segment */
```

Description

Determines if a memory segment is possibly corrupted.

This function returns the dirty-flag of the memory segment specified by handle. The dirty-flag of a memory segment is set before each write call to a segment and reset afterwards. Thus, the return of this function can be used to determine if a write operation has been interrupted, for instance due to a power-failure.

Returns:

- 1
  dirty-flag is set
- 0
  dirty-flag is not set
- NVRAM_ERR_INIT
  The NVRAM library is not initialized
- NVRAM_ERR_HANDLE
  The specified handle is invalid

---

nvram_get_pointer( )

Synopsis

```
void *nvram_get_pointer(
    nvram_handle handle)                    /* In: handle of a memory segment */
```

Description

Return a pointer to the start of a memory segment.

This function returns a pointer to the start of a NVRAM memory segment in order to enable direct memory access to the segment.

NOTE!

Direct memory access circumvents all boundary checking and checksum testing performed by the regular nvram_read() and nvram_write() functions, thus making it possible to accidentally overwrite the content of other segments. Furthermore an indication if the data is corrupted as otherwise provided by nvram_get_dirty_flag() does not exist. Finally, calling nvram_realloc() may invalidate the returned pointer without notification. Due to these reasons, it is strongly advised against using this function for regular NVRAM access; it shall only be used in exceptional cases.

Returns a pointer to the memory segment or NULL in case of an error

---

nvram_info( )

Synopsis

```
int32_t nvram_info(
    nvram_dev_info_t* dev_info,          /* Out: Overall device information */
    nvram_seg_info_t* seg_info_array)    /* Out: Segment information. */
```

Description

Fetch detailed information on the NVRAM device.

Fills the specified buffers dev_info and seg_info_array with information from the NVRAM device.
dev_info and seg_info_array have to be allocated / freed by the calling function.
seg_info_array points to an array of the size NVRAM_MAX_NUM_SEGS.

Returns:

- >=0
  Success, number of entries in the seg_info array
- NVRAM_ERR_INIT
  The NVRAM library is not initialized

---

nvram_init( )

Synopsis

```
int32_t nvram_init(
    void )
```

Description

This function shall not be used.
It issues an SE-log if called.

---

nvram_malloc( )

Synopsis

```
nvram_handle nvram_malloc(
    const char *name,                /* In: name for the memory segment */
    int32_t     version,             /* In: version for the memory segment */
    size_t      size)                /* In: size of the memory segment */
```

Description

Allocate a non-volatile memory segment.

When allocating a memory segment for the first time a management data block is initialized for the new segment and the segment data is cleared (set to 0). In addition, the new segment is marked as dirty. Consecutive calls to this function with the same parameters for name, version and size will return a handle of the previously allocated memory segment provided the segment has not been released using nvram_free() in the meantime.

Returns:

- >=0
  Success, a valid handle to the memory segment is returned

- NVRAM_ERR_INIT
  The NVRAM library is not initialized
- NVRAM_ERR_NAME
  The specified name is invalid
- NVRAM_ERR_VERSION
  A memory segment with the specified name already exists but its version does not match the specified version.
- NVRAM_ERR_SIZE
  A memory segment with the specified name already exists but its size does not match the specified size.
- NVRAM_ERR_NOMEM
  Not enough memory available to fulfill the request

---

nvram_read( )

Synopsis

```
int32_t nvram_read(
    nvram_handle   handle, /* In: handle of a memory segment */
    unsigned char *buffer, /* In: pointer to read buffer */
    size_t         length, /* In: number of bytes to read */
    int32_t        offset) /* In: starting offset within the memory segment */
```

Description

Read data from a memory segment.

Reads length bytes starting at the specified offset from the memory segment specified by handle into the specified buffer. The buffer must be large enough to receive up to length bytes.

Returns:

- >=0
  Success, number of bytes read
- NVRAM_ERR_INIT
  The NVRAM library is not initialized
- NVRAM_ERR_HANDLE
  The specified handle is invalid item NVRAM_ERR_SEGMENT
  Segmentation error; the specified offset is larger than the segment size

---

nvram_realloc( )

Synopsis

```
nvram_handle nvram_realloc(
    nvram_handle handle,     /* In: handle of the memory segment to modify */
    int32_t      version,    /* In: new version for the memory segment */
    size_t       size)       /* In: new size for the memory segment */
```

Description

Change the size and/or version of an already existing memory segment.

This function can be used to change the size or version of a memory segment that has already been allocated before. A successfully completed reallocation resets the dirty flag of the segment. If the request could not be fulfilled an appropriate error code will be returned and the existing segment will not be modified.

Newly allocated memory will not be initialized.

Returns:

- >=0
  Success, the handle passed as first parameter is returned
- NVRAM_ERR_INIT
  The NVRAM library is not initialized
- NVRAM_ERR_HANDLE
  The specified handle is invalid
- NVRAM_ERR_NOMEM
  Not enough memory available to fulfill the request

---

nvram_reset( )

Synopsis

```
int32_t nvram_reset(
    void)
```

Description

Reset the NVRAM device.

This function clears all of the NVRAM management data, thus resetting the device. All information stored in the device will be lost.

Returns NVRAM_OK on success or NVRAM_ERR_INIT if the NVRAM library is not initialized.

---

nvram_sc_get( )

Synopsis

```
int32_t nvram_sc_get(
    const char *name,                          /* In:  System counter name */
    int64_t    *p_ctr )                        /* Out: 64-bit counter value */
```

Description

Get the value of a system counter.

---

nvram_segment_info( )

Synopsis

```
nvram_handle nvram_segment_info(
    const char* name,                /* In:  Name specifying the segment */
    nvram_seg_info_t* seg_info)      /* Out: Segment information */
```

Description

Fetch detailed information about a memory segment.

Fills the buffer seg_info with details on the NVRAM memory segment with the specified name. This function can be used to identify NVRAM memory segments that are already in use.
seg_info has to be allocated / freed by the calling function.
If seg_info is NULL the data structure will not be filled with the requested details. If name can be found the return value will be a valid handle

Returns:

- >=0
  Success, handle of the segment specified by name
- NVRAM_ERR_HANDLE
  The specified name is invalid

- NVRAM_ERR_INIT
  The NVRAM library is not initialized

---

nvram_write( )

Synopsis

```
int32_t nvram_write(
    nvram_handle         handle,         /* In: handle of a memory segment */
    const unsigned char *buffer,         /* In: pointer to write buffer */
    size_t               length,         /* In: number of bytes to write */
    int32_t              offset) /* In: starting offset within the memory segment */
```

Description

Write data to a memory segment.

Writes length bytes from the specified buffer into the memory segment specified by handle starting at the specified offset. A successfully completed write operation resets the dirty flag of the segment.

Returns:

- >=0
  Success, number of bytes written
- NVRAM_ERR_INIT
  The NVRAM library is not initialized
- NVRAM_ERR_HANDLE
  The specified handle is invalid
- NVRAM_ERR_SEGMENT
  Segmentation error; the specified offset is larger than the segment size

2.20 OS - Operating System Service

The CSS Operating System service provide API functions for basic real time operating system functionality.

Basic functionality includes:

- Application management for loading and starting applications
- Device management for reseting of the device
- Task management for activating and control tasks
- Interrupt management for enabling and disabling interrupts
- Semaphore management for synchronization and mutual exclusion
- Time management for setting, reading and convert system time
- Message queue management for task communication

OS Time API

The OS time service includes API functions for setting and getting the system time and the local time and for setting the local time zone. The OS time service essentially handles three different time formats:

ANSI	Local time. Mostly used for user input/output.
POSIX	Portable high resolution system time format (64 bits).
TIMEDATE48	Non-portable compact system time format (48 bits).

Below are the C declarations used for the different time formats. ANSI time is used to store local time while the other time formats hold system time.

```
typedef struct ansi_time
{
    INT32 tm_sec;           seconds after the minute - [0, 59]
    INT32 tm_min;           minutes after the hour   - [0, 59]
    INT32 tm_hour;          hours after midnight     - [0, 23]
    INT32 tm_mday;          day of the month         - [1, 31]
    INT32 tm_mon;           months since January     - [0, 11]
    INT32 tm_year;          years since 1900
    INT32 tm_wday;          days since Sunday        - [0, 6]
    INT32 tm_yday;          days since January 1     - [0, 365]
    INT32 tm_isdst;         Daylight Saving Time flag
} OS_STR_TIME_ANSI;

typedef struct posix_time
{
    UINT32 sec;             seconds
    UINT32 nanosec;         nanoseconds
} OS_STR_TIME_POSIX;

typedef struct OS_STR_TIMEDATE48
{
    UINT32 seconds;
    UINT16 ticks;           updated 60 times/second, resolution 1/65536 sec
} OS_TIMEDATE48;
```

The functions for getting and setting time the POSIX time are os_c_get and os_c_set. These functions affect the system time of the device. The OS time service also provides functions for converting between the different time formats:

From	To	Function
ANSI time	System time in seconds	os_c_mktime
System time in seconds	ANSI time	os_c_localtime

os_appl_load( )

Synopsis

```
int os_appl_load(
    const char* p_file_name,    /* In:  Application file name */
    char* p_ref_name,           /* In:  Application name */
    void** pp_module)           /* Out: Module ID */
```

Description

This routine loads an application (object module) pointed out by p_file_name.

The application must be defined in the device configuration as deferred started, i.e. the application is left unloaded and is not started by the RTS system starter.

Parameter p_file_name is the file name including the path relative to the system working directory, e.g. the directory where the objects.lst file is located.

Parameter p_ref_name must be the application name defined in the device configuration, e.g. the exp_app_name field in the structure AS_APPLICATION.

The object module id is passed in the output parameter pp_module to be used by the application if the object module shall be unloaded.

The symbol table for the application, which is linked together with the application (object module), is added to the CSS symbol handler in the call to this function.

Returns OK if success or ERROR in the following cases:

- a parameter in the call is incorrect
- the application can not be found in the device configuration
- the application name p_ref_name is too long
- the application or its dependencies is not correct
- the symbol table can not be found

os_appl_start( )

Synopsis

```
int os_appl_start(
    char* p_appl_name)                          /* In: Application file name */
```

Description

This function starts an application at runtime.

The application must be declared as deferred started in the device configuration. This means that the RTS system starter ignores to load, initialize and start this application and leaves it to the application to decide when to load and start it.

Starting the application includes starting each and every task defined in the application configuration.

Parameter p_appl_name must be the application name defined in the device configuration, e.g. the exp_app_name field in the structure AS_APPLICATION.

Returns OK if the function succeeded to start the application else ERROR if the device configuration can not be found or if an application task can not be started.

os_as_task_period( )

Synopsis

```
INT16 os_as_task_period(
```

```
    char* task_name,            /* IN : Name of a cyclic task */
    UINT16* configurated_time,  /* OUT: Configured cycle time of the task */
    UINT16* calculated_time  )  /* OUT: The cycle time the task will have
                                        when it is started.
                                        The output values will be set to zero
                                        if the task is not AS_TT_CYCL. */
```

Description

This function can be used to check the configured cycle time of a cyclic task with task name specified in task_name.

The douplicated parameters configured_time and calculated_time is for compatible reason and will have the same value.

Returns OK on success or ERROR if the task is not a cyclic task or any argument is NULL.

---

os_c_get( )

Synopsis

```
INT16 os_c_get(
    OS_STR_TIME_POSIX  *p_time)  /* Out: System time */
```

Description

This function gets the system time in POSIX format using the VxWorks clock ID CLOCK_REALTIME.

The system time is stored in the buffer p_time.

Returns OK on success or ERROR if the system clock could not be read or p_time is NULL.

---

os_c_get_monotonic( )

Synopsis

```
INT16 os_c_get_monotonic(
    OS_STR_TIME_POSIX  *p_time)  /* Out: System time */
```

Description

This function gets the system time in POSIX format using the VxWorks clock ID CLOCK_MONOTONIC.

The system time is stored in the buffer p_time.

Returns OK on success or ERROR if the system clock could not be read or p_time is NULL.

---

os_c_gmtime( )

Synopsis

```
INT16 os_c_gmtime(
    UINT32            sec,                    /* In:  System time */
    OS_STR_TIME_ANSI  *p_ansi_time)           /* Out: UTC time */
```

Description

The function converts calendar time specified in input_sec into broken-down time in the buffer output_ANSI_time. After the conversion the broken-down time is in UTC time zone.

Returns ERROR if p_ansi_time is NULL else returns OK.

---

os_c_localtime( )

Synopsis

```
INT16 os_c_localtime(
    UINT32            sec,                    /* In:  System time */
    OS_STR_TIME_ANSI  *p_ansi_time)           /* Out: Local time */
```

Description

The function converts calendar time specified in input_sec into broken-down time in the buffer output_ANSI_time. After the conversion the broken-down time is in local time zone.

Returns ERROR if p_ansi_time is NULL else returns OK.

---

os_c_mktime( )

Synopsis

```
INT16 os_c_mktime(
    const OS_STR_TIME_ANSI  *p_ansi_time,         /* In:  Local time */
    UINT32                  *p_sec)               /* Out: System time */
```

Description

The function convert broken-down time (ANSI) into system time in seconds.

The time that should be converted is stored at p_ansi_time and the converted seconds is stored at p_sec.

Returns OK on success ERROR if the conversion fails or input args are NULL.

---

os_c_res_get( )

Synopsis

```
INT16 os_c_res_get(
    INT32  *p_res_ms,               /* Out: system time resolution in ms */
    INT32  *p_tics_per_sec)         /* Out: system time resolution in ticks */
```

Description

This function gets the resolution of the system clock in both milliseconds and ticks per second.
The value at p_tics_per_sec has better accuracy than the value at p_res_ms. If the value at p_tics_per_sec is greater than 1000 (hz) then the value at p_res_ms is always set to 1 (ms).

Returns OK on success or ERROR if the system clock resolution could not be computed or if either argument ir NULL.

---

os_c_set( )

Synopsis

```
INT16 os_c_set(
    const OS_STR_TIME_POSIX  *p_time)  /* In: System time (UTC) */
```

Description

The function set the system clock to a specified time in POSIX format. If a RTC (Real Time Clock) is present it is set to the same time.

Returns OK on success or ERROR if:

- the system clock can not be set
- time is outside the supported range
- nanosecond value is out of range
- p_time is NULL

---

os_hi_reset( )

Synopsis

```
void os_hi_reset(
    void)
```

Description

This function sets the reset reason to OS_RESET_ERROR and resets the device.

os_hi_shutdown( )

Synopsis

```
void os_hi_shutdown(
    UINT16 restart_mode)                          /* In:  The reset mode */
```

Description

This function shuts down all applications and restarts the device.

The mode that the device shall start up in is set by restart_mode.

The supported modes are:

OS_RESET_DVS	
OS_RESET_IDLE	
OS_RESET_RUN	
OS_RESET_ERROR	
OS_RESET_REINIT	
OS_RESET_SHUTDOWN	For CCU-O2, the same behaviour as for OS_RESET_ERROR

os_i_disable( )

Synopsis

```
INT16 os_i_disable(
    INT32  *p_lockKey)      /* Out: Pointer to Lock-out Key */
```

Description

This function disables CPU interrupts.

The p_lockKey contains an architecture-dependent lock-out key representing the interrupt level prior to the call. The p_lockKey is used when the interrupt is enabled.

Returns OK always.

os_i_enable( )

Synopsis

```
INT16 os_i_enable(
    INT32  lockKey)         /* In: Lock key */
```

Description

This function re-enables CPU interrupts that has been disabled.

The parameter lockKey is an architecture-dependent lock-out key provided by os_i_disable().

Returns OK always.

os_load_module( )

Synopsis

```
int os_load_module(
    int                     fd,             /* In: File to read from */
    int                     symFlag)        /* In: Load no symbols = 0, otherwise load all
```

Description

This function loads an object module from a file into memory at runtime.

This function is used to load modules in a system where RAM test is enabled. The RAM test is verifying the loaded modules by calculating its checksum in the text section. This test cannot execute at the same time as the operating system loads the module because it adjusts the test section checksum. By using this function the RAM tests does not execute while the system loads the module.

Parameter symFlag is specifying if all or no symbols shall be loaded into the system symbol table.

Returns OK if the function succeeded to load the object module else ERROR if the the function cannot read the file, there is not enough memory or the file format is illegal.

---

os_q_create( )

Synopsis

```
INT16 os_q_create(
    INT32 maxMsgs,              /* IN : Total amount of messages to be queued */
    INT32 maxMsgLength,         /* IN : Max. number of bytes per message */
    INT32 options,              /* IN : OS_MSG_Q_FIFO or OS_MSG_Q_PRIORITY */
    INT32* msg_q_id)            /* OUT: Message queue id reserved for this queue */
```

Description

This routine creates a message queue capable of holding up to maxMsgs messages, each up to maxMsgLength bytes long. The routine sets the message queue identifier (msg_q_id) used to identify the created message queue in all subsequent calls to messages queue functions.

The queue can be created with the following options:

OS_MSG_Q_FIFO	queue pended tasks in FIFO order
OS_MSG_Q_PRIORITY	queue pended tasks in priority order

Returns OK on success ERROR if not enough memory or called from an interrupt service routine or msg_q_id is NULL.

---

os_q_delete( )

Synopsis

```
INT16 os_q_delete(
    INT32 msgQId)               /* IN: Message queue id */
```

Description

This function delete a specified message queue from the system.

All task pending on os_q_receive() or os_q_send() for this msgQId will unblock and return ERROR. When this function returns, msgQId is no longer valid.

Returns OK on success or ERROR in the following cases:

- invalid message queue id
- called from interrupt service routine

---

os_q_receive( )

Synopsis

```
INT16 os_q_receive(
    INT32 msg_q_id,             /* IN : Message queue id */
    INT8* buffer,               /* IN : Pointer to the receive buffer */
    UINT maxNBytes,             /* IN : Max. size of the receive buffer */
```

```
    INT32 timeout,              /* IN : Timeout in ms    > 0
                                        OS_NO_WAIT         0
                                        OS_WAIT_FOREVER   -1 */
    INT32* rcv_size)            /* OUT: Total number of bytes received */
```

Description

This routine receives a message from the message queue defined by msg_q_id. The received message is copied into the specified buffer, which is maxNBytes in length. If the message is longer than maxNBytes, the remainder of the message is discarded (no error indication is returned).

The parameter rcv_size is the total number of received bytes.

The timeout parameter specifies the number of milliseconds to wait for a message to be sent to the queue, if no message is available when the function is called.

The timeout parameter can also have the following special values:

OS_NO_WAIT	return immediately, whether a message has been received or not
OS_WAIT_FOREVER	never time out

Returns OK on success or ERROR in the following cases:

- invalid message queue id
- either buffer or rcv_size is NULL
- message queue deleted while calling task was pended
- no free buffer space when OS_NO_WAIT timeout was specified
- timeout occurred while waiting for buffer space
- message buffer size exceeds limit
- called from interrupt service routine

---

os_q_send( )

Synopsis

```
INT16 os_q_send(
    INT32 msg_q_id,             /* IN: Message queue id */
    INT8* buffer,               /* IN: Pointer to the buffer to send */
    UINT32 nBytes,              /* IN: Number of bytes to send */
    INT32 timeout,              /* IN: Timeout in ms    > 0
                                       OS_NO_WAIT         0
                                       OS_WAIT_FOREVER   -1 */
    INT32 priority)             /* IN: OS_MSG_PRI_NORMAL or OS_MSG_PRI_URGENT */
```

Description

This routine sends the message in buffer of length nBytes to the message queue msg_q_id. If any tasks are already waiting to receive messages on the queue, the message will immediately be delivered to the first waiting task. If no task is waiting to receive messages, the message is saved in the message queue and if a task has previously registered to receive events from the message queue, these events are sent in the context of this call. This may result in the unpending of the task waiting for the events.

The timeout parameter(timeout) specifies the number of milliseconds to wait for free space if the message queue is full.

The timeout parameter can also have the following special values:

OS_NO_WAIT	return immediately, even if the message has not been sent.
OS_WAIT_FOREVER	never time out

The priority parameter specifies the priority of the message being sent.

The possible values are:

OS_MSG_PRI_NORMAL	normal priority, add the message last of the queued messages
OS_MSG_PRI_URGENT	urgent priority, add the message first of the queued messages

Returns OK on success or ERROR in the following cases:

- invalid message queue id
- buffer is NULL
- message queue deleted while calling task was pended
- no free buffer space when OS_NO_WAIT timeout was specified
- timeout occurred while waiting for buffer space
- message length exceeds limit
- called from interrupt service routine with non-zero timeout

---

os_s_delete( )

Synopsis

```
INT16 os_s_delete(
    INT32 semId)                /* IN: Semaphore identifier */
```

Description

This routine terminates and deallocates any memory associated to the semaphore with identifier semId.

All tasks pending on the semaphore will unblock and return ERROR.

Returns OK on success or ERROR in the following cases:

- invalid semaphore id
- called from interrupt service routine for a mutex semaphore

---

os_s_give( )

Synopsis

```
INT16 os_s_give(
    INT32 semId)                /* IN: Semaphore identifier */
```

Description

This routine performs the give operation on a specified semaphore with identifier semId.

Depending on the type of semaphore, the state of the semaphore and of the pending tasks may be affected.

Returns OK on success or ERROR in the following cases:

- invalid semaphore id
- called from interrupt service routine for a mutex semaphore
- task not owner of the mutex semaphore

---

os_s_take( )

Synopsis

```
INT16 os_s_take(
    INT32 semId,                /* IN: Semaphore identifier */
    INT32 timeout)              /* IN: OS_NO_WAIT       0
                                       OS_WAIT_FOREVER -1
                                       time in ms     > 0 */
```

Description

This function performs the take operation on a the semaphore with identifier semId.

Depending on the type of semaphore, the state of the semaphore and the calling task may be affected.

The parameter timeout is specified as timeout in milliseconds. If the timeout is set to OS_NO_WAIT the function will not wait for the semaphore to be free. The timeout could also be set to OS_WAIT_FOREVER which means that the task will wait forever for the specified semaphore to be available.

Returns OK on success or ERROR in the following cases:

- invalid semaphore id
- task timed out
- called from interrupt service routine for a mutex semaphore

---

os_sb_create( )

Synopsis

```
INT16 os_sb_create(
    INT32 options,                /* IN : OS_SEM_Q_FIFO or OS_SEM_Q_PRIORITY */
    OS_SEM_B_STATE initialState, /* IN : OS_SEM_EMPTY or OS_SEM_FULL */
    INT32* id)                    /* OUT: Semaphore identifier */
```

Description

This routine allocates and initializes a binary semaphore.

Tasks can be queued on a priority basis or a first-in-first-out basis. These options are OS_SEM_Q_PRIORITY and OS_SEM_Q_FIFO, respectively. The task que handling is set by the parameter options.

The initial state of the semaphore is set by initialState to either OS_SEM_EMPTY or OS_SEM_FULL.

The parameter id holds the identifier of the semaphore to be used in give, take or delete operations on the semaphore.

Returns OK on success or ERROR if the creation of the semaphore fails due to out of memory or id is NULL.

---

os_sm_create( )

Synopsis

```
INT16 os_sm_create(
    INT32 options,              /* IN : OS_SEM_Q_FIFO, OS_SEM_Q_PRIORITY,
                                       OS_SEM_DELETE_SAFE, OS_SEM_INVERSION_SAFE */
    INT32* id)                  /* OUT: Semaphore identifier */
```

Description

This function allocates and initializes a mutual-exclusion semaphore.

The semaphore state is initialized to full.

The parameter options supports the following values:

OS_SEM_Q_FIFO	queue pended tasks on a first-in-first-out basis
OS_SEM_Q_PRIORITY	queue pended tasks on the basis of their priority
OS_SEM_DELETE_SAFE	protect a task that owns the semaphore from
-	unexpected deletion
OS_SEM_INVERSION_SAFE	protect the system from priority inversion.
-	With this option,the task owning the semaphore will
-	execute at the highest priority of the tasks pended
-	on the semaphore, if it is higher than its current priority.
-	This option must be accompanied by the OS_SEM_Q_PRIORITY
-	queuing mode.

These values can be OR'ed together to build the desired option.

The identifier of the created semaphore is stored in id to be used in give, take or delete operations on the semaphore.

Returns OK on success or ERROR if the creation of the semaphore fails due to out of memory or an invalid option is specified or id is NULL.

---

os_start_mode_get( )

Synopsis

```
int os_start_mode_get(
    void )
```

Description

This function returns the current start mode of CSS.

Returns:
OS_START_MODE_IDLE if CSS is running in IDLE mode.
OS_START_MODE_RUN if CSS is running in RUN mode.

---

os_t_delay( )

Synopsis

```
INT16 os_t_delay(
    UINT32 delay_time_ms)              /* IN: Time to delay in milliseconds */
```

Description

The function causes the calling task to relinquish the CPU for delay_time_ms milliseconds.

This is commonly referred to as manual rescheduling, but is also useful when waiting for some external condition.

A task delay means that the task gives up the CPU and is ready to excecute the next system tick after the delay time has expired.
The system tick rate is 60 Hz. i.e. one tick every 16.7 ms.
If the shortest possible delay is wanted and the delay time is set to zero, it could be that the actual delay is up to 17 ms. This must be considered when calling this routine from supervised very fast cyclic tasks.

Returns OK on success or ERROR if called from interrupt level.

---

os_t_delete( )

Synopsis

```
INT16 os_t_delete(
    void)
```

Description

This function deletes the calling task.

Returns OK on success or ERROR if the task not could be found in the CSS task handler.

---

os_t_event_raise( )

Synopsis

```
INT16 os_t_event_raise(
    char *task_name)                    /* IN: Task name to send event to */
```

Description

The function send an event to the task specified.

Returns OK on success or ERROR if the task_id is invalid or the function is called from an interrupt service routine or task_name is NULL.

---

os_t_event_wait( )

Synopsis

```
INT16 os_t_event_wait(
    void)
```

Description

This function puts the calling task into a wait state for an event.

The task will wait forever for the event to occur.

Returns OK on success or ERROR if the function is called from an interrupt service routine.

---

os_t_name( )

Synopsis

```
void os_t_name(
    char** task_name)            /* OUT: The name of the calling task */
```

Description

This routine gets the task name for the calling task.
If task_name is NULL then this function does nothing.
If the task does not exist the function outputs NULL in task_name.

---

os_t_spawn( )

Synopsis

```
INT16 os_t_spawn(
    const char appl_name[],     /* IN : Application name connected to the task */
    INT32 appl_type,            /* IN : Type of application AS_TYPE_AP_C,
                                        AS_TYPE_AP_TOOL */
    char task_name[],           /* IN : Name of the task */
    UINT8 priority,             /* IN : Priority of the task */
    INT32 stack_size,           /* IN : Size of the stack */
    FUNCPTR entry_pt,           /* IN : Start adress of the task */
    INT32 argc,                 /* IN : No of arguments to the function (Max 10.) */
    INT32 argv[],               /* IN : The argument list */
    UINT32* task_id)            /* OUT: VxWorks task id */
```

Description

This routine creates a new task with the specified priority.

The allowed priorities for application tasks are 10 - 249 and 254.

The parameter appl_name is the name of the application that the task shall be connected to.

The type of applicatiton (appl_type) may be AS_TYPE_AP_C or AS_TYPE_AP_TOOL.

The name of the task is sent as parameter task_name and it is used to identify the task.

The only resource allocated to a spawned task is a stack of a specified size stack_size. The stack size is recommended to be not less than 4 kByte for any task.

The entry address entry_pt is the address of the task body routine, that will be called when the task is started.

The parameter argc is the number of calling arguments that is sent to the task body function.

The parameter argv is the argument vector, that is a pointer to the list of calling arguments to the task body function.

The identifier of the task is returned as task_id.

Returns OK on success or ERROR in the following cases:

- type of application is invalid
- the taks name is not unique
- the stack size is zero
- application name is too long
- illegal priority for an application task
- the function is called from an interrupt service routine
- not enough memory to allocate for the stack
- can not add the task to the CSS task handler
- can not add the task to the CSS exception hander
- task_id is NULL

os_time_zone_set( )

Synopsis

```
int os_time_zone_set(
    const char  *zone_name_str,   /* In: Name of the time zone (3 chars) */
    const char  *offset_str,      /* In: String with offset from UTC in minutes */
    const char  *dst_start_str,   /* In: Daylight savings start MMDDHH */
    const char  *dst_end_str)     /* In: Daylight savings end MMDDHH */
```

Description

Set the time zone and optionally the DST of the local time.

Returns OK if successfull.
Returns ERROR on error.

os_timedate_48_clock_get( )

Synopsis

```
INT16 os_timedate_48_clock_get(
    OS_TIMEDATE48  *p_timedate48)                  /* Out: System time */
```

Description

This function gets the current time from the system clock. The time is stored in p_timedate48.

Returns OK on success or ERROR if the system clock could not be read or p_timedate48 is NULL.

os_timedate_48_clock_set( )

Synopsis

```
INT16 os_timedate_48_clock_set(
    OS_TIMEDATE48   timedate48)                   /* In: System time to set */
```

Description

This function sets the time according to the value in the timdedate48 struct.

Returns OK on success or ERROR if:

- the system clock can not be set

- time is outside the supported range
- nanosecond value is out of range

---

os_ver( )

Synopsis

```
void os_ver(
    int *p_version,                         /* Out: CSS version */
    int *p_release,                         /* Out: CSS release */
    int *p_update,                          /* Out: CSS update  */
    int *p_build)                           /* Out: CSS build  */
```

Description

This Function can be used to get the version of CSS.

The CSS version is in form of:

V(version).R(release).U(update).B(build).

---

os_verify_file( )

Synopsis

```
int os_verify_file(
    const char* file_name)                      /* In: Application file name */
```

Description

This routine verifies the integrity of a file by asking the DVS service.

Returns OK if the file exists and either the integrity check is disabled or the integrity of the file is OK.
Returns ERROR if the file does not exists or the integrity check fails.

2.21 PAR - External Parameter Service

Overview

The PAR service in CSS provides support to create a parameterized configuration. It is possible to modify the actual values of parameter data used by the applications on a device without affecting the applications. This way it is possible to create generic or standard applications that could be modified through parameters without changing the actual application.

At startup of the system the PAR service traverses the default parameter mapping directory for files. Each file found is regarded as active. All active files are parsed according to the XML format for a mapping file. Every parameter value file pointed out by the mapping file is parsed according to the XML format for a value file.

Not until all parameter files are successfully parsed the PAR service is started and the API function is available. The parsing of the files does not include validation of the parameter name and its value. If a parameter should be missing or its value is missing it will be handled when the API function is called.

To be able to retrieve a value for a parameter the PAR service provides an API function to the application. Before an MTPE application is started this API function is called by the MTPE application framework and the parameter gets its initial value according to the settings in MTPE.

Enum used for the data types in PAR API function call:

```
typedef enum TAG_TCMS_C_FUNCTION_INTERFACE_TYPES
{
    TCMS_BOOL               = 100,
    TCMS_BOOLEAN8           = 101,
    TCMS_BYTE               = 102,
    TCMS_WORD               = 103,
    TCMS_DWORD              = 104,
    TCMS_FLOAT32            = 105,
    TCMS_FLOAT64            = 106,
    TCMS_INT8               = 107,
    TCMS_INT16              = 108,
    TCMS_INT32              = 109,
    TCMS_INT64              = 110,
    TCMS_UINT8              = 111,
    TCMS_UINT16             = 112,
    TCMS_UINT32             = 113,
    TCMS_UINT64             = 114,
    TCMS_MT_UNICODE16       = 115,
    TCMS_MWT_BYTE           = 116,
    TCMS_MWT_WORD           = 117,
    TCMS_MWT_DWORD          = 118,
    TCMS_MWT_ANALOG         = 119,
    TCMS_MWT_BCD4           = 120,
    TCMS_MWT_BIFRACT200     = 121,
    TCMS_MWT_BOOL           = 122,
    TCMS_MWT_BOOLEAN2       = 123,
    TCMS_MWT_DATE           = 124,
    TCMS_MWT_DATE_AND_TIME  = 125,
    TCMS_MWT_DINT           = 126,
    TCMS_MWT_ENUM4          = 127,
    TCMS_MWT_FIXED          = 128,
    TCMS_MWT_INT            = 129,
    TCMS_MWT_REAL           = 130,
    TCMS_MWT_SINT           = 131,
    TCMS_MWT_STRING         = 132,
    TCMS_MWT_TIME           = 133,
    TCMS_MWT_TIME_OF_DAY    = 134,
    TCMS_MWT_TIMEDATE48     = 135,
    TCMS_MWT_UDINT          = 136,
    TCMS_MWT_UINT           = 137,
    TCMS_UNIFRACT           = 138,
```

```
    TCMS_MWT_USINT          = 139
}TCMS_C_FUNCTION_INTERFACE_TYPES;
```

PAR Configuration

The parameter configuration files consist of two types of files, the parameter mapping file and the value file. Each parameter mapping file contains definition of all parameters and their corresponding parameter value file. Each mapping file can contain reference to one or several parameter value file. Each parameter value file contains the actual value of the parameter and their corresponding names.

The parameter value files could be modified without changing the configuration. Parameter mapping files must be placed in the default directory "tcms_parameter_mapping" on the root of the files system. No other files should be stored at this location.

For more information about the configuration of the parameters refer to: [CSSCRM]

---

par_get_value( )

Synopsis

```
INT16 par_get_value(
    char const * const Application_Name,       /* In:  Name of the application,
                                                       must not be NULL  */
    char const * const Parameter_Name,         /* In:  Name of the parameter,
                                                       must not be NULL */
    TCMS_C_FUNCTION_INTERFACE_TYPES Data_type, /* In:  Data type on parameter */
    void const * const Min_value,              /* In:  Pointer to min value,
                                                       type depending on data type,
                                                       could be NULL */
    void const * const Max_value,              /* In:  Pointer to max value,
                                                       type depending on data type,
                                                       could be NULL */
    void *Par_value)                           /* Out: Pointer to value,
                                                       type depending on data type,
                                                       must not be NULL */
```

Description

This function provides the value of an external parameter to the application. It searches the parameter XML files for the parameter value. The parameter is associated with an application according to the parameter in the call. The value is interpreted according to the data type specified in the call. The value of the parameter can be specified as decimal or hexadecimal in the XML file. The user could specify a minimum and a maximum limit for the parameter value. If any of the parameters for min and max value are NULL the limit check is omitted. If the limits are specified they are interpreted according to the data type in the call.
Supported data types are:
- BOOL (0)
- WORD (1)
- UINT8 (2)
- INT8 (3)
- UINT16 (4)
- INT16 (5)
- UINT32 (6)
- INT32 (7)
- FLOAT32 (8)

The function returns PAR_OK (0) if the value can be found and is within the limits.
The function returns PAR_ERROR (-1) in the following cases:
- A pointer in the call to this function is NULL
- The XML parsing of the set of external parameter files failed
- The PAR service is not started
- The value in the XML file can not be interpreted according to the data type

The function returns PAR_NAME_ERROR (-2) if the parameter cannot be found.
The function returns PAR_TYPE_MISMATCH (-3) if the data types mismatches and no conversion can be done.

The function returns PAR_RANGE_CONSTRAINT (-4) if the value is outside the min or max limit.
The function returns PAR_TYPE_VIOLATION (-5) if the data type is inconsistent with the type specified in the value file.

2.22 PPP - CSS Point-to-Point Protocol Support Service

Overview

The PPP CSS Support Service exposes an API to configure and initialize a Point-to-Point Protocol (PPP) network interface over a point-to-point serial link.

The PPP service supports the following physical serial ports and serial interfaces on a VCU-C2:

- Serial A, RS232 or RS485 half duplex, baud rates: 9600, 19200, 38400 and 115200
- Serial B, RS422 or RS485 half or full duplex, baud rates: 9600, 19200, 38400 and 115200

The network interface created by the PPP service will be named: ppp0.

---

ppp_css_config( )

Synopsis

```
INT16 ppp_css_config(
        INT16        deviceDuplex,   /* In: Serial device and duplex */
        INT16        baudrate,       /* In: Baud rate */
        const char * localAddress)   /* In: Local address in dotted decimal notation
                                             e.g. "1.1.1.1" */
```

Description

This function is used for configuring a PPP connection. It configures which device to be used, half or full duplex, baud rate and the local address for the PPP connection.

A PPP connection must be configured before the PPP connection is started with a call to the function ppp_css_up().

The parameter deviceDuplex configures the device name and duplex for the PPP connection.

Valid values for deviceDuplex are:

PPP_SERIAL_A_RS232	Serial A and RS232 full duplex
PPP_SERIAL_A_HALF_DUPLEX	Serial A and RS485 half duplex
PPP_SERIAL_B_HALF_DUPLEX	Serial B and RS485 half duplex
PPP_SERIAL_B_FULL_DUPLEX	Serial B and RS422 full duplex
PPP_SERIAL_USB_HALF_DUPLEX	Serial USB and RS485 half duplex
PPP_SERIAL_USB_FULL_DUPLEX	Serial USB and RS422 full duplex

The parameter baudrate sets up the baud rate.

Valid values for baudrate are:

PPP_BAUDRATE_9600	Baud rate is set to 9600
PPP_BAUDRATE_19200	Baud rate is set to 19200
PPP_BAUDRATE_38400	Baud rate is set to 38400
PPP_BAUDRATE_115200	Baud rate is set to 115200

The parameter localAddress is the local address in dotted decimal notation e.g. ”10.1.1.255”

This function immediately returns an error if the system has not yet discovered the device. This usually happens when an external USB to serial adapter is used for PPP connection. USB to serial adapter may take several seconds to be discovered and mounted in system. In such situation, it is recommended to call this function repeatedly until success.

Returns OK if successful.
Returns ERROR if:
- The parameter deviceDuplex is invalid.
- The system has not yet discovered device.
- Fails to configure device, baudrate or localAddress.
- Fails to initialize interface.

---

ppp_css_down( )

Synopsis

```
INT16 ppp_css_down(
```

```
    void)
```

Description

This routine stops the ongoing PPP connection.

Returns OK or ERROR

---

ppp_css_up( )

Synopsis

```
INT16 ppp_css_up(
    void)
```

Description

This routine starts a PPP connection. Before this function is called the PPP interface should have been configured by the use of the function ppp_css_config().

Returns OK or ERROR

2.23 PS - Periodic Scheduler Service

The CSS Periodic Scheduler service manages the cyclic tasks in the system.

The service provides a number of features, e.g. cyclic scheduling, overrun supervision, delayed execution and rescheduling of overrunning task.

The resolution of the task cycles is one millisecond.

A periodic task shall be constructed this way:

```
xxx periodic_task(...)
{
    ps_add(...);
    while (!terminate_condition)
    {
        ps_wait(...);
        task code
        ...
    }
    ps_delete(...);
}
```

If a task shall be added to the CSS Periodic Scheduler it must be spawned using the CSS os_t_spawn() API function. The reason is that CSS prepares the TCB (Task Control Block) with information used by the CSS Periodic Scheduler.

A programming example how to spawn a periodic task can look like this:

```
xxx task_spawn_function(...)
{
    os_t_spawn(MY_APP_NAME, AS_TYPE_APP_C, MY_TASK_NAME,
               MY_TASK_PRIO, MY_STACK_SIZE,
               (FUNCPTR)periodic_task, my_arg_num,
               my_argv, &my_task_id);
}
```

---

ps_add( )

Synopsis

```
INT16 ps_add(
    UINT32 cycle_time,                /* In:  Cycle time in ms */
    UINT32 first_delay_time,          /* In:  Delay to first schedule in ms */
    UINT16 wdog_delay,                /* In:  Number of allowed overruns */
    UINT16 *p_timer_index)            /* Out: PS timer ID */
```

Description

This function adds the calling task to the CSS Periodic Scheduler.

The first time ps_wait is called the task will be suspended for first_delay_time + cycle_time.

The parameter wdog_delay specifies the number of consecutive overruns that is allowed for the task. If one or several overruns has occurred the overrun counter will be reset the next time the task meets its deadline. If the task does not meet its deadline within the specified wdog_delay the device will be shutdown and restarted. The parameter will be overriden by the global watchdog flag in the device configuration.

The identifier for the cyclic task to be used in the call to ps_wait is output in p_timer_index.

Returns OK on success or ERROR if the task has not been spawned by os_t_spawn() or if there are no timers left.

---

ps_cycle_time_get( )

Synopsis

```
void ps_cycle_time_get(
    UINT16 * cycle_time)        /* Out: Cycle time in ms */
```

Description

Returns the base cycle time in cycle_time which is always 1 (one) ms.

ps_cycles_get( )

Synopsis

```
void ps_cycles_get(
    UINT32 * activation_count)                    /* Out: Nbr of PS tics */
```

Description

This function outputs the accumulated Periodic Scheduler tick counter in activation_count.

Overflow in the data type is not considered.

ps_delete( )

Synopsis

```
INT16 ps_delete(
    UINT16 timer_index)                           /* In:  PS timer ID */
```

Description

Deletes the task and removes it from the CSS Periodic Scheduler.

The Periodic Scheduler timer allocated for the task is then free to be used by another task.

The parameter timer_index is the identifier for the cyclic task in the Periodic Scheduler and is not valid after a call to this function.

A controlled termination of a periodic task shall be constructed as:

```
xxx periodic_task(...)
{
    ps_add(...);
    while(!terminate_condition)
    {
        ps_wait(...);
        task code
        ...
    }
    ps_delete(...);
}
```

It is possible to delete a periodic task from another task by calling this function with the correct timer_index. The task associated with timer_index will be suspended and deleted before it is removed from the Periodic Scheduler.

Returns OK on success or ERROR if the timer_index is not associated with a cyclic task.

ps_time_since_last_execution( )

Synopsis

```
INT16 ps_time_since_last_execution(
    UINT32 * cycle_time)        /* Out: The time [ms] since last start of
                                        execution. */
```

Description

Returns ERROR, the function is not implemented.

---

ps_wait( )

Synopsis

```
INT16 ps_wait(
    UINT16 timer_index)                          /* In:  PS timer ID */
```

Description

This function shall be called before the actual task code (see description in header). The the deadline for the next cycle will be calculated and the task will be prepared for its next execution. When the task has finished its execution and calls this function, the Periodic Scheduler will take care of any task overrun that might have occurred.

If a task overrun has occurred the following actions will be taken:

The maximum number of overruns is reached	the device is shut down
The maximum number of overruns is not reached	the task is scheduled to its next cycle
Reschedule is specified in the device configuration	the task is resumed at once

The parameter timer_index is the identifier for the cyclic task in the Periodic Scheduler from the call to ps_add().

If a callback function is connected to the task by the function pshook_register() the function will be called before the task is scheduled for its next cycle.

Returns OK always.

---

pshook_info_get( )

Synopsis

```
INT16 pshook_info_get(
    UINT32  hook_id,                             /* In: The hook id */
    UINT32 *p_user_info)                         /* Out: The returned user info */
```

Description

This function gets a user-defined, hook specific, info field. This info field can be set by calling pshook_info_set().

Returns OK on success or ERROR if hook_id is not a valid call-back identifier or if the parameter p_user_info is NULL.

---

pshook_info_set( )

Synopsis

```
INT16 pshook_info_set(
    const char task_name[],                      /* In: Name of the hooked task */
    UINT32     hook_id,                          /* In: The hook id */
    UINT32     user_info)                        /* In: The user info to set */
```

Description

This function sets a user-defined, hook specific, info field. This info field can be queried by calling pshook_info_get().

Returns OK on success or ERROR if the task is not a CSS cyclic task or if hook_id is not a valid call-back identifier for the task or if parameter hook_id is 0.

---

pshook_register( )

Synopsis

```
INT16 pshook_register(
    const char task_name[],  /* In: Task name of associated call-back task */
```

```
    PS_FUNC_PTR p_hook_ftn,  /* In: Call-back function reference */
    UINT32 * hook_id)        /* Out: Call-back function identifier */
```

Description

This function registers a call-back function for a cyclic task.

The registered function is called every time the periodic scheduler calls pshook_run with the exttcb of task task_name.

The periodic scheduler call-back functionality can be used to synchronize and add code execution to a cyclic task.

The parameter task_name is the task to which the call-back shall be associated.

The parameter p_ps_func is a reference to the call-back function of type PS_FUNC_PTR.

```
typedef void (*PS_FUNC_PTR) (const char* task_name, UINT32 hook_id);
```

The identifier for the registred call-back function is stored in hook_id.

Returns OK on success or ERROR if the task is not a CSS cyclic task or in case of out of memory or if any of the parameters hook_id or p_hook_ftn is NULL.

---

pshook_unregister( )

Synopsis

```
INT16 pshook_unregister(
    const char task_name[], /* In: Task name of associated call-back task */
    UINT32 hook_id)         /* In: Call-back function identifier */
```

Description

This function unregisters a registered call-back function for a periodic task.

The parameter task_name specifies the task associated with the call-back function.

The parameter hook_id is the identifier for the call-back function to unregister.

Returns OK on success or ERROR if the task is not a CSS cyclic task or if hook_id is not a valid call-back identifier for the task or if parameter hook_id is 0.

2.24 SE - System Event Log Service

Overview

CSS provides an event log for CSS itself and with CSS integrated products, e.g. TCN, NVRAM manager and IPTCom, to store error conditions. The system event log is stored in NVRAM memory, using two circular buffers of approx. 10 entries each. When a buffer becomes full the oldest entry will be over written.
Entries are also stored to file as they arrive.

Two generic strategies are defined for SE-Log messages:

SE_WARNING	used for non-critical messages
SE_ERROR	used for critical messages

Log Traversal:

The functions se_get_newest_index and se_get_entry are the preferred API to be used for traversing a specific strategy list. Function se_get_entry retrieves the full message from the log.

The following functions are retained for legacy purposes. They are deprecated and are not recommended for new applications:

The functions se_last_struct_get and se_indx_struct_get can be used to traverse a specific strategy list.
The functions se_last_get and se_prev_get can be used to traverse both strategy lists (starting with the most recent log entry).

Log Traversal Caveats:

- Due to limitations in the legacy API functions only one SE list traversal may be active at any time. API functions se_get_newest_index, se_last_struct_get and se_last_get all start a new traversal.
- Do NOT modify the index! They are not simple integer indices.
- Only those entries stored in the NVRAM can be traversed using these three function groups.
- There exists no API function for traversing the contents of the files used for long-term storage.

The CallBack Mechanism:

Applications may register a function that CSS calls when a new entry is inserted into the log. The call-back is called from an internal CSS task which runs at low priority and thus does not disturb normal operation.

The call-back must be registered by an init task in order to make sure that all entries are reported to the call-back function, even after a restart of the device.

If the internal NVRAM buffer overruns before the entry is reported to the application then the entry will not be reported and no report of the lost entry is provided.

The API function to register a call-back function is se_callback_add.
The old API function se_callback_reg is deprecated and is retained for legacy applications. It is not recommended for new applications.
A call-back can be removed by calling the se_callback_del function.

_du_hamster_put( )

Synopsis

```
void _du_hamster_put(
    const CHAR* p_file_id,      /* In: File Identification*/
    UINT16 line_nr,             /* In: Line Number in File */
    INT16 strategy,             /* In: Strategy
                                       - CONTINUE
                                       - RESET
                                       - HALT
                                       - SHUTDOWN
                                       - STALL */
    INT32 count,                /* In: Number of subsequent params */
    ... )                       /* In: Parameters */
```

Description

Note: This function is deprecated.
New applications should use se_put instead.
This function is added for backwards compatibility only. Do NOT use this function for new applications.

_du_hamster_text( )

Synopsis

```
void _du_hamster_text(
    const CHAR* p_file_id,      /* In: File Identification */
    UINT16 line_nr,             /* In: Line Number in File */
    INT16 strategy,             /* In: Strategy
                                       - CONTINUE
                                       - RESET
                                       - HALT
                                       - SHUTDOWN */
    const CHAR* fmt,            /* In: Format expression,
                                       built like printf()
                                       format strings ... */
    ... )                       /* In: Parameters in order as
                                       described in format string. */
```

Description

Note: This function is deprecated.
New applications should use se_put instead.
This function is added for backwards compatibility only. Do NOT use this function for new applications.

se_callback_add( )

Synopsis

```
int se_callback_add(
    LOG_CB_FUNC_PTR func,                   /* In: Function pointer */
    int            *p_cb_id)                /* Out: Id of the call-back */
```

Description

This function registers a call-back function to be called when a new message is logged.

The call-back function, takes one argument: a pointer to a struct of type LOG_ENTRY.
In this struct the SE-Log message is passed to the application through the call-back call.

Following the defines and typedef used in the call-back function:

```
#define LOG_MESSAGE_LENGTH  250

typedef struct STR_LOG_DATA
{
    UINT32             seconds;
    INT16              strategy;
    UINT32             id;
    char               message[LOG_MESSAGE_LENGTH];
} LOG_ENTRY;
```

seconds	the time-stamp for the message
strategy	the type of message, SE_ERROR or SE_WARNING
id	the identity for the message
message	the buffer where the message is put

An identity for the registred call-back function is returned in cb_id, to be used when deleting it.

Returns OK on success or ERROR if failed to register the call-back function.

se_callback_del( )

Synopsis

```
int se_callback_del(
     int id)                    /* In: Call-back register id */
```

Description

Delete the registered call-back function. The call-back function is selected by the parameter id.

Returns OK on success or ERROR if failed to delete the call-back function.

se_callback_reg( )

Synopsis

```
int se_callback_reg(
    SEFUNCPTR func,             /* In:  Call-Back function pointer */
    int      *id)               /* Out: Id of the call-back */
```

Description

Note: This function is deprecated.
Use se_callback_add instead in new applications.

se_get_entry( )

Synopsis

```
int se_get_entry(
    INT16        strategy,                  /* In:    SE_ERROR or SE_WARNING */
    int         *p_index,                   /* InOut: Entry index */
    LOG_ENTRY   *p_se_data )                /* Out:   The log entry */
```

Description

This function reads the SE-Log entry indicated by *p_index. The message is stored in the buffer se_data.

The p_index argument is the requesting index whose entry should be returned in *p_se_data. Upon exit the value has been modified to indicate the next, older, entry to get.

The type of message of interest is defined in strategy, SE_WARNING or SE_ERROR.

Returns OK on success or ERROR if no message is available or any parameter in the call is unvalid.

se_get_first( )

Synopsis

```
int se_get_first(
    INT16        strategy,                  /* In:  SE_ERROR or SE_WARNING */
    int         *p_index,                   /* Out: Entry index */
    LOG_ENTRY   *p_se_data )                /* Out: The log entry */
```

Description

Gets the oldest logged message from the SE-Log.
The message is returned in the buffer at p_se_data.

The type of message of interest is specified in strategy: SE_WARNING or SE_ERROR.

Upon exit *p_index holds the index for the next oldest stored message. The data of which can be used as input to se_get_next.

Returns OK on success or ERROR if the last message could not be read.

se_get_newest_index( )

Synopsis

```
int se_get_newest_index(
    INT16        strategy,                  /* In:  SE_ERROR or SE_WARNING */
    int         *p_index )                  /* Out: Index to newest entry */
```

Description

Sets index to the newest logged entry in the SE-Log.

The type of message of interest is specified in strategy: SE_WARNING or SE_ERROR.

Upon exit *p_index holds the index for the newest stored entry which can be used as input to se_get_entry.

Returns OK on success or ERROR if the last message could not be read.

se_get_next( )

Synopsis

```
int se_get_next(
    INT16        strategy,                  /* In:    SE_ERROR or SE_WARNING */
    int         *p_index,                   /* InOut: Entry index */
    LOG_ENTRY   *p_se_data )                /* Out:   The log entry */
```

Description

This function reads the SE-Log message indicated by idx. The message is stored in the buffer se_data.

The index parameter is the requesting index that should be returned in *p_se_data. Upon exit the value has been modified to indicate the next structure to get.

The type of message of interest is defined in strategy, SE_WARNING or SE_ERROR.

Returns OK on success or ERROR if no message is available or any parameter in the call is unvalid.

se_indx_struct_get( )

Synopsis

```
int se_indx_struct_get(
    int  idx,                   /* In:  Requesting index */
    int *p_next_idx,            /* Out: Next Index  */
    int  strategy,              /* In:  Strategy of message (SE_ERROR or SE_WARNING) */
    SE_LOG_DATA* p_se_data)     /* Out: Struct containing message log */
```

Description

Note: This function is deprecated.
New applications should use se_get_newest_index/se_get_entry instead.
This function is added for backwards compatibility only.
Do NOT use this function for new applications.

This function reads the SE-Log message indicated by indx. The message is stored in the buffer se_msg.

The idx parameter is the index of the entry that should be returned as se_msg. Range 0..(max-1). The function se_last_struct_get can be used to get (max-1).

The index for the next (earlier) message, if any, is returned in p_next_idx.

The type of message of interest is defined in strategy, SE_WARNING or SE_ERROR.

Returns OK on success or ERROR if no message is availabe or any parameter in the call is invalid.

se_info( )

Synopsis

```
void se_info(
    INT16* p_entry_count,       /* Out: The number of entries */
    INT16* p_current_index)     /* Out: Next free index */
```

Description

This function prints the internal log header to standard output.

Sets the output arguments to zero.

se_last_struct_get( )

Synopsis

```
int se_last_struct_get(
    int           strategy,     /* In:  SE_ERROR or SE_WARNING */
    SE_LOG_DATA *p_se_data,     /* Out: Struct containing message log */
    int          *p_idx)        /* Out: Entry index */
```

Description

Note: This function is deprecated.
New applications should use se_get_newest_index/se_get_entry instead.
This function is added for backwards compatibility only.
Do NOT use this function for new applications.

Gets the last logged message from the SE-Log. The message is returned in the buffer se_msg.

The type of message of interest is defined in strategy, SE_WARNING or SE_ERROR.

Upon exit p_idx holds the index for the stored message. The value of which can be used as input to se_indx_struct_get.

Returns OK on success or ERROR if the last message could not be read.

se_log_last_get( )

Synopsis

```
INT16 se_log_last_get(
    char  *log_entry,           /* Out: The SE-Log message */
    INT16 *p_idx)               /* Out: The index of the message */
```

Description

Note: This function is deprecated.
New applications should use se_get_newest_index/se_get_entry instead.
This function is added for backwards compatibility only.
Do NOT use this function for new applications.

This function gets the last logged message as a string in the buffer log_entry.

Upon exit p_idx holds the index for the returned message.

Returns OK on success otherwise ERROR.

se_log_prev_get( )

Synopsis

```
int se_log_prev_get(
    char  *log_entry,           /* Out:   The SE-Log message */
    INT16 *p_idx)               /* InOut: The index of the message */
```

Description

Note: This function is deprecated.
New applications should use se_get_newest_index/se_get_entry instead.
This function is added for backwards compatibility only. Do NOT use this function for new applications.

This function gets the previous SE-Log message in the buffer log_entry.

Upon entry p_idx hold the index for the current message. Upon exit p_idx holds the index for the returned message.

Returns OK on success otherwise ERROR.

---

se_log_text( )

Synopsis

```
int se_log_text(
    const char *p_file_id,      /* In: Pointer to FILE NAME */
    UINT16      line_nr,        /* In: Line number of the logged event */
    const char *fmt,            /* In: The formated text string */
    ...)
```

Description

Note: This function is deprecated.
New applications should use se_put instead.
This function is added for backwards compatibility only. Do NOT use this function for new applications.

---

se_put( )

Synopsis

```
void se_put(
    INT16        strategy,      /* In: Strategy SE_ERROR or SE_WARNING */
    UINT32       id,            /* In: A unique id of the message */
    const char  *message,       /* In: The message string */
    ...)
```

Description

This function writes a messsage into the SE-Log.

The message can be a format message string with a list of arguments (...).

The user should identify the creator of the message by assigning different identities to different applications or modules.

2.25 SYM - Symbol Handler Service

The symbol handling in CSS is done by the CSS Symbol Service.

The Symbol Service is use to translate symbolic names into address and type information. In fact, entire C-line expressions may be parsed by CSS in order to traverse array elements and structure members.

Symbols are stored in symbol tables, and symbol tables can be loaded and unloaded at runtime using Symbol Service API or Monitor commands. Memory addresses referenced by symbols can be displayed and modified using CSS monitor commands.

The symbol table is generated automatically when the application is built by MTPE. MTPE links the application with the generated symbol table. This way the handling of symbol tables is completely transparent to the user (It Just Works).

If the application is not built using MTPE then the PC utility program sym_gen can be used to create a C-file containing the symbol table.
This file must then be compiled and linked to the application before the final file is inserted into a DLU file.
The utility sym_gen is installed when installing MTPE.

The CSS symbol table itself can be displayed and traversed using CSS monitor commands.

For more details on Symbol Service monitor commands refer to: [MON]

Normally the application symbol table is added to the CSS Symbol Service automatically when application is loaded by the CSS application loader.

The maximum number of application symbol tables in the system is 32.

The following definitions and typedefs are used throughout the CSS Symbol Handler API:

```
#define SYM_SYMID_NONE      -1
#define SYM_SYMTYPE_NONE    -1
#define SYM_SYMTAB_ALL      -1
#define SYM_SYMTAB_NONE     -2

#define SYMTAB_NAME_MAX     128

typedef int SYM_TABLE_ID;
typedef int SYM_TYPE_ID;
typedef int SYM_ID;

typedef enum
{
    sym_integer_type,
    sym_char_type,
    sym_pointer_type,
    sym_float_type,
    sym_bool_type,
    sym_array_type,
    sym_struct_union_class_type,
    sym_member_type,
    sym_enum_type,
    sym_enum_value_type,
    sym_number_of_types
} SYM_TYPE_CODE;

typedef struct STR_SYM_TYPE_INFO
{
    SYM_TYPE_CODE type_id;
    UINT32 size_in_byte;
    union
    {
        struct
        {
            BOOL is_signed;
        } is_integer_type;
```

```
        struct
        {
            BOOL is_signed;
        } is_char_type;
        struct
        {
            SYM_TYPE_ID element_type_id;
            UINT32 element_count;
        } is_array_type;
        struct
        {
            SYM_TYPE_ID member_type_id;
            const char *member_name;
            UINT32 member_offset;
            UINT32 member_size;
        } is_member_type;
        struct
        {
            INT32 enum_value;
            const char *enum_name;
        } is_enum_value_type;
        struct
        {
            SYM_TYPE_ID ref_type_id;
        } is_pointer_type;
    } type_attrib;
} SYM_TYPE_INFO;
```

sym_add_table( )

Synopsis

```
int sym_add_table(
    void         *p_table,          /* In:  Symbol table pointer */
    const char   *table_name,       /* In:  Name of symbol table */
    SYM_TABLE_ID *p_table_id)       /* Out: Symbol table id */
```

Description

This function adds a symbol table to the CSS Symbol Handler.

The symbol table to add is defined by p_table. table_name is the name that the symbol table shall be referenced with in the CSS Symbol Handler.

If the symbol table p_table already exist in the CSS symbol Handler table_name will be the valid table name.

The identity for the symbol table is stored in p_table_id.

Returns OK if the symbol table is successfully added.
Returns ERROR in the following cases:

- p_table is NULL or not a reference to a symbol table.
- the version of the symbol table is not correct.
- the maximum number of symbol tables in the system is reached.

sym_del_table( )

Synopsis

```
int sym_del_table(
    SYM_TABLE_ID table_id)          /* In: Symbol table id */
```

Description

This function removes a symbol table from the Symbol Handler.

The parameter table_id is the identifier for the symbol table.

Returns OK if the symbol table was successfully removed. Returns ERROR in the following cases:

- table_id is invalid
- table_id is not an identifier for an added symbol table

---

sym_find_by_name( )

Synopsis

```
int sym_find_by_name(
    SYM_TABLE_ID  table_id,         /* In:  Symbol table id */
    const char   *sym_string,       /* In:  Name of symbol */
    SYM_TABLE_ID *p_table_id,       /* Out: Symbol table id */
    void*        *p_sym_address,    /* Out: Address of symbol */
    SYM_TYPE_ID  *p_type_id)        /* Out: Symbol type id */
```

Description

This function searches for a symbol with a specific name in a symbol table.

The parameter table_id specifies which symbol table to search.
If table_id is set to SYM_SYMTAB_ALL the function searches all symbol tables in the system.

The parameter sym_string specifies the symbol to look for.

On success the p_table_id contains the pointer to the first symbol table where the symbol is found.

The parameter p_sym_address is the address to the found symbol.

The type of the found symbol is stored in p_type_id.

Returns OK if the symbol could be found.
Returns ERROR in the following cases:

- p_table_id is NULL
- sym_address is NULL
- sym_string is NULL
- p_type_id is NULL
- table_id is invalid
- the symbol can not be found
- the symbol is not unique

---

sym_find_in_table( )

Synopsis

```
int sym_find_in_table(
    const char   *table_name,       /* In:  Name of symbol table */
    const char   *sym_string,       /* In:  String contain name of symbol */
    SYM_TABLE_ID *p_table_id,       /* Out: Symbol table id */
    void*        *p_sym_address,    /* Out: Address of symbol */
    SYM_TYPE_ID  *type_id)          /* Out: Symbol type id */
```

Description

This function finds a symbol in a specific symbol table.
The table_name specifies which symbol table to search.
The sym_string specifies the symbol to look for.

The id for the symbol table in which the symbol was found is stored in p_table_id.
The address referenced by the symbol is stored at p_sym_address.
In type_id is an identifier for the type of the symbol stored.

Returns OK if the symbol was found.
Returns ERROR in the following cases:

- any argument is NULL.
- the table cannot be found.
- the symbol table does not exist
- the symbol cannot be found
- the symbol is not unique

---

sym_find_symbol_table( )

Synopsis

```
int sym_find_symbol_table(
    const char* file_name,           /* In:  Module or application name */
    void** pp_symbol_table )         /* Out: Symbol table */
```

Description

This function finds the CSS Symbol table associated with an application.

The name of the application object module file is defined in file_name.

A reference to the symbol table is output in pp_symbol_table.

Returns OK if the symbol table is found.
Returns ERROR in the following cases:

- any argument is NULL
- the object module can not be found
- the object module is not linked with a valid symbol table
- the file_name exceeds SYMTAB_NAME_MAX

---

sym_get_first_member( )

Synopsis

```
int sym_get_first_member(
    SYM_TABLE_ID table_id,           /* In: Symbol table id */
    SYM_TYPE_ID  type_id,            /* In: Symbol type id */
    SYM_TYPE_ID *p_member_id)        /* Out: First member id */
```

Description

This function retrieves the first member of a struct, union, class or enum in a symbol table.

The parameter table_id specifies which symbol table to search.

The parameter type_id is the type identifier for which the first member shall be retrieved.

On success, the identifier for the first member is stored in p_member_id.

Returns OK on success.
Returns ERROR in the following cases:

- the table_id is invalid
- the type_id is not a struct, union, class or enum
- p_member_id is NULL

---

sym_get_first_symbol( )

Synopsis

```
int sym_get_first_symbol(
    SYM_TABLE_ID  table_id,          /* In:  Symbol table id */
    SYM_ID       *p_sym_id)          /* Out: Symbol id */
```

Description

This function retrieves the first symbol in a symbol table.

The parameter table_id specifies which symbol table to search.

On success, the symbol identifier for the first symbol is stored in p_sym_id.

Returns OK on success.
Returns ERROR in the following cases:

- the table_id is invalid
- the table_id is an invalid symbol table identifier

---

sym_get_first_table_id( )

Synopsis

```
int sym_get_first_table_id(
    SYM_TABLE_ID* table_id)          /* Out: First symbol table id */
```

Description

This function retrieves the first symbol table from the Symbol Handler.

If no symbol tables exist in the system, SYM_SYMTAB_NONE is stored in table_id.

Returns OK if the first symbol table could be found.
Returns ERROR if table_id is NULL.

---

sym_get_next_member( )

Synopsis

```
int sym_get_next_member(
    SYM_TABLE_ID table_id,           /* In:  Symbol table id */
    SYM_TYPE_ID  member_id,          /* In:  Current member id */
    SYM_TYPE_ID *p_next_member_id)   /* Out: Next member id */
```

Description

This function retrieves the next member relative a specific member of a struct, union, class or enum in a symbol table.

The parameter table_id specifies which symbol table to search.

The parameter member_id is the member identifier for which the next member shall be searched for.

If member_id is the last member there are no more members and p_prev_member_id will be set to SYM_SYMTYPE_NONE. This is not a failure so the function returns OK.

On success, the identifier for the next member is stored in p_next_member_id.

Returns OK on success.
Returns ERROR in the following cases:

- the table_id is invalid
- the member_id is not a struct, union, class or enum
- p_next_member_id is NULL

---

sym_get_next_symbol( )

Synopsis

```
int sym_get_next_symbol(
    SYM_TABLE_ID  table_id,          /* In:  Symbol table id */
    SYM_ID        sym_id,            /* In:  Current symbol id */
    SYM_ID       *p_next_sym_id)     /* Out: Next symbol id */
```

Description

This function retrieves the next symbol relative a specific symbol in a symbol table.

The parameter table_id specifies which symbol table to search.

The parameter sym_id is the symbol identifier for which the next symbol shall be searched for.

If sym_id is the last symbol in the symbol table there are no more symbols and p_next_sym_id will be set to SYM_SYMID_NONE. This is not a failure so the function returns OK.

Returns OK on success.
Returns ERROR in the following cases:

- table_id is invalid
- sym_id is invalid
- p_next_sym_id is NULL

---

sym_get_next_table_id( )

Synopsis

```
int sym_get_next_table_id(
    SYM_TABLE_ID  table_id_in,       /* In:  Current symbol table id */
    SYM_TABLE_ID *table_id)          /* Out: Next symbol table id */
```

Description

This function retrieves the next symbol table relative a specified symbol table from the Symbol Handler.

The parameter table_id_in is the symbol table identifier from which the next symbol table shall be searched.
If table_id_in is less than zero, the first symbol table identifier is stored in table_id.

If table_id_in is greater or equal to the maximum nuber of symbol tables in the system, SYM_SYMTAB_NONE is stored in table_id.
If table_id_in is the last symbol table in the system, SYM_SYMTAB_NONE is stored in table_id.

Returns OK if the first symbol table could be found.
Returns ERROR if table_id is NULL.

---

sym_get_prev_member( )

Synopsis

```
int sym_get_prev_member(
    SYM_TABLE_ID table_id,           /* In:  Symbol table id */
    SYM_TYPE_ID  member_id,          /* In:  Current member id */
    SYM_TYPE_ID* p_prev_member_id)   /* Out: Previous member id */
```

Description

This function retrieves the previous member relative a specific member of a struct, union, class or enum in a symbol table.

The parameter table_id specifies which symbol table to search.

The parameter member_id is the member identifier for which the previous member shall be searched for.

If member_id is the first member there are no previous members and p_prev_member_id will be set to SYM_SYMTYPE_NONE. This is not a failure so the function returns OK.

On success, the identifier for the previous member is stored in p_prev_member_id.

Returns OK on success.
Returns ERROR in the following cases:

- the table_id is invalid
- the member_id is not a struct, union, class or enum
- p_prev_member_id is NULL

---

sym_get_prev_symbol( )

Synopsis

```
int sym_get_prev_symbol(
    SYM_TABLE_ID table_id,           /* In:  Symbol table id */
    SYM_ID       sym_id,             /* In:  Current symbol id */
    SYM_ID      *p_prev_sym_id)      /* Out: Previous symbol id */
```

Description

This function retrieves the previous symbol relative a specific symbol in a symbol table.

The parameter table_id specifies which symbol table to search.

The parameter sym_id is the symbol identifier for which the previous symbol shall be searched for.

If sym_id is the first symbol in the symbol table there are no previous symbols and p_prev_sym_id will be set to SYM_SYMID_NONE. This is not a failure so the function returns OK.

Returns OK on success.
Returns ERROR in the following cases:

- table_id is invalid
- sym_id is invalid
- p_prev_sym_id is NULL

sym_get_sym_address( )

Synopsis

```
int sym_get_sym_address(
    SYM_TABLE_ID table_id,           /* In:  Symbol table id */
    SYM_ID       sym_id,             /* In:  Symbol id */
    void*       *p_sym_address)     /* Out: Address of symbol */
```

Description

This function retrieves the address for a symbol in a symbol table.

The parameter table_id specifies which symbol table to search.

The parameter sym_id is the symbol identifier for which the address shall be retrieved.

A pointer to the address of the symbol is returned in p_sym_address.

Returns OK on success.
Returns ERROR in the following cases:

- the table_id is invalid
- sym_id is invalid
- pp_sym_address is NULL

sym_get_sym_name( )

Synopsis

```
int sym_get_sym_name(
    SYM_TABLE_ID table_id,           /* In: Symbol table id */
    SYM_ID       sym_id,             /* In: Symbol id */
    const char* *p_sym_name)         /* Out: Symbol name */
```

Description

This function retrieves the name for a symbol in a symbol table.

The parameter table_id specifies which symbol table to search.

The parameter sym_id is the symbol identifier for witch the name shall be retrieved.

A pointer to the name of the symbol is returned in p_sym_name.

Returns OK on success.
Returns ERROR in the following cases:

- the table_id is invalid
- sym_id is invalid
- p_sym_name is NULL

sym_get_sym_type( )

Synopsis

```
int sym_get_sym_type(
    SYM_TABLE_ID table_id,           /* In:  Symbol table id */
    SYM_ID       sym_id,             /* In:  Symbol id */
    SYM_TYPE_ID *p_type_id)          /* Out: Symbol type id */
```

Description

This function retrieves the type for a symbol in a symbol table.

The parameter table_id specifies which symbol table to search.

The parameter sym_id is the symbol identifier for which the type shall be retrieved.

The type identifier is stored in p_type_id.

Returns OK on success.
Returns ERROR in the following cases:

- table_id is invalid
- sym_id is invalid
- p_type_id is NULL

---

sym_get_table_id( )

Synopsis

```
int sym_get_table_id(
    const char   *table_name,        /* In:  Name of the symbol table */
    SYM_TABLE_ID *p_table_id)        /* Out: Symbol table id */
```

Description

This function retrieves the symbol table identifier for the symbol table defined by table_name.

The symbol table identifier is stored in p_table_id.

Returns OK if the symbol table is found.
Returns ERROR if:

- any argument is NULL.
- the table cannot be found.

---

sym_get_table_name( )

Synopsis

```
int sym_get_table_name(
    SYM_TABLE_ID table_id,           /* In:  Symbol table id */
    const char** p_table_name)       /* Out: Name of table */
```

Description

This function retrieves the name of a symbol table based on a symbol table identifier.

A pointer to the name of the symbol table is stored in p_table_name.

Returns OK if the symbol table name can be found.
Returns ERROR if table_id is invalid.

---

sym_get_type_info( )

Synopsis

```
int sym_get_type_info(
    SYM_TABLE_ID   table_id,         /* In:  Symbol table id */
    SYM_TYPE_ID    type_id,          /* In:  Symbol type id */
    SYM_TYPE_INFO *p_sym_type_info)  /* Out: Symbol type info */
```

Description

This function retrieves the type information for a symbol in a symbol table.

The retrieved type information is used together with other API functions when traversing the symbol table and to be able to interpret all kinds of symbol types.

The parameter table_id specifies which symbol table to search.

The parameter type_id is the type identifier for which the information shall be retrieved.

The type information is stored in p_sym_type_info.

Type code and type size in bytes are always retrieved.
The following additional information is supplied depending on the type code:

Type code	Additional information
sym_array_type	Element type id, Element count
sym_bool_type	-
sym_char_type	Sign
sym_enum_type	-
sym_enum_value_type	Value, Name
sym_float_type	-
sym_integer_type	Sign
sym_member_type	Member type id, Member name, Member offset in bits, Member size in bits
sym_pointer_type	Referenced type id
sym_struct_union_class_type	-

Returns OK on success.
Returns ERROR in the following cases:

- p_sym_type_info is NULL
- the table_id is invalid
- type_id is invalid

---

sym_load_table( )

Synopsis

```
int sym_load_table(
    const char*   file_name,     /* In:  Name of file */
    const char*   table_name,    /* In:  Name of symbol table - May be NULL */
    BOOL          verify,        /* In:  Not implimented */
    SYM_TABLE_ID* p_table_id)    /* Out: Symbol table identifier */
```

Description

This function both loads and adds a symbol table to the CSS Symbol Handler.

The file_name is the name of the file that shall be loaded and table_name is the name that the symbol table shall be referenced with in the CSS Symbol Handler.
If table_name is a NULL pointer or points to an empty string, a default name will be the name of the symbol table.
The default name will be constructed from the file name appended with ”_symbolic_table_”.

The parameter verify specifies if the checksum of the file shall be verified. This functionality is not supported.

Returns OK if the symbol table was loaded.
Returns ERROR in the following cases:

- Either file_name or p_table_id is NULL
- the file name is invalid

- the file is not found
- the symbol table module could not be loaded
- the symbol table could not be added to the CSS symbol table
- the file name exceeds SYMTAB_NAME_MAX
- the reference for the symbol table in the file can not be found
- the version of the symbol table is not correct

---

sym_set_alias( )

Synopsis

```
int sym_set_alias(
    SYM_TABLE_ID table_id,           /* In: Table id */
    const char  *alias)              /* In: Alias to set */
```

Description

This function sets an alias for a symbol table.
The parameter table_id is the identifier for the symbol table.
The parameter alias defines the alias name to set.
When a new alias is set for a symbol table, the former alias is removed.

A name for a symbol table (alias) must be unique for all symbol tables in the system. The name cannot be empty.

Returns OK if the alias name could be set.
Returns ERROR in the following cases:

- table_id is invalid
- alias id NULL or empty
- alias is not unique amongst all symbol tables

---

sym_unload_table( )

Synopsis

```
int sym_unload_table(
    SYM_TABLE_ID table_id)           /* In: Symbol table id */
```

Description

This function removes and unloads a previous loaded symbol table from the Symbol Handler.

The parameter table_id is the identifier for the symbol table.

Returns OK if the symbol table is sucessfully unloaded.
Returns ERROR in the following cases:

- the table_id is invalid
- the symbol table was never added to the Symbol Handler
- the symbol table has never been loaded by the system

2.26 TCN - Train Communication Network Service

This service implements the Train Communication Network (TCN).

For a detailed description refer to: [TCNPD], [TCNMD] and [TCNBA].

2.27 TRDP - Train Real-time Data Protocol Service

Overview

The TRDP service is responsible for the initiating and starting of the IEC61375 (TRDP) communication stack.

The TRDP service will only try to initiate and start the TRDP communication stack if there is a TRDP configuration file located on the device, and if the device is about to enter RUN mode.
The TRDP configuration file should be placed at the following location: /app0/trdp_config.xml.
In case the device is forced to enter IDLE mode there will be no attempt in trying to start the TRDP communication stack.

CSS only support TRDP communication on one Ethernet interface, the primary Ethernet interface.

TRDP is not supported in EOS. Neither the TRDP service nor the IEC61375 (TRDP) communication stack has been integrated in CSS for EOS.

For more information on how to use TRDP refer to: [CCUAG] and [TRDPUM].

---

trdp_get_app_handle( )

Synopsis

```
TRDP_HANDLE_T trdp_get_app_handle(
    UINT8 interfaceNumber) /* In: Ethernet interface number. */
                           /*     The only supported value is 0, */
                           /*     i.e. primary Ethernet interface */
```

Description

Function for retrieving the TRDP application handle.

During the initialization of the TRDP communication stack a so-called application handle is created. This handle is required by subsequent calls to other TRDP API's. For more information refer to: [TRDPUM].

Returns:
- Pointer to the application handle
- NULL in case of error

The function may fail during the following conditions:
- Wrong interfaceNumber provided
- The TRDP stack is not yet fully initiated and started
- The initialization and start of the TRDP stack failed

Note: This function is not supported in EOS.

2.28 TS - Time Sync Service

CSS provides a Time Synchronization service that, together with the Device Configuration structure AS_TIME_SYNC_INIT makes it possible to synchronize devices over the MVB bus.

The device can be configured to be a Time Sync Master or a Time Sync Slave on the MVB bus.

Note that a device does not have to be a MVB bus master to be a MVB time sync master.
Note that the Time Sync service is only running when the device is in RUN mode, because it needs a Device Configuration and process data variables to function properly.

The data in the MVB port consists of four variables, like this:

Type	Content
UINT32	The time in seconds since jan 1, 1970.
UINT16	The number of 1/65536 parts of the current second.
INT16	The number of minutes to compensate for the local time zone.
INT16	The number of minutes to compensate for DST.

For more information about the CSS Time Sync functionality refer to: [CSSCRM]

---

time_sync_get( )

Synopsis

```
INT16 time_sync_get(
    OS_TIMEDATE48 *sync_UTC_time,              /* Out: UTC Time */
    OS_TIMEDATE48 *sync_zone_time,             /* Out: Local time */
    OS_TIMEDATE48 *get_offset_time,            /* Out: Offset time */
    INT8          *get_offset_sign,            /* Out: Offset sign */
    INT16         *get_time_zone,              /* Out: Time zone offset */
    INT16         *get_daylight )              /* Out: DST offset */
```

Description

This function is used for retrieving the MVB Time Synchronization parameters in different formats.

The synchronized time (UTC) is stored in the buffer sync_UTC_time in TIMEDATE48 format.

The synchronized time in local time zone format is stored in the buffer sync_zone_time.
This time is calculated from the local time, the offset_time and the time_zone and daylight in TIMEDATE48 format.

The absolute time difference between the local time and the UTC time is stored in the buffer get_offset_time in TIMEDATE48 format.

```
typedef struct OS_STR_TIMEDATE48
{
    UINT32 seconds;
    UINT16 ticks;
} OS_TIMEDATE48;
```

The sign of the offset is stored in get_offset_sign and the meaning is as following:

Value	Meaning
0	Local time (get_offset_time) is higher than source time
1	Local time (get_offset_time) is lower than source time

Time difference in minutes with respect to UTC is stored in the parameter get_time_zone.
The time difference is typically -60 for Central European Time.

Daylight saving time difference in minutes with respect to the standard time is stored in the parameter get_daylight.

Returns OK if the arguments are not NULL and time sync is active.
Returns ERROR if:
- Any argument to this function is NULL.
- Time Synchronization was not configured in the Device Configuration.
- It was not possible to retrieve the device time.

time_sync_set( )

Synopsis

```
INT16 time_sync_set(
    OS_TIMEDATE48 offset_time,                  /* In:  Offset time */
    INT8          offset_sign,                  /* In:  Offset sign */
    INT16         time_zone,                    /* In:  Time zone offset */
    INT16         daylight)                     /* In:  DST offset */
```

Description

This function sets the MVB Time Synchronization parameters.

The use of this function is required when the time_source in AS_TIME_SYNC_INIT is set to AS_TS_API, but it can also be used to override the configuration specified in the Device Configuration.

If the DC parameter ”time_source” is set to AS_TS_API then handling of the master time is external to CSS, e.g. in a time synchronization application receiving the time from a radio clock or train time master over WTB or Ethernet.
CSS has to be informed when and how to update the time, and this is done using this function which overrides any information written in the Device Configuration.

After a reset of the time master the parameters from the Device Configuration is used.
This might cause a problem, so if ”time_source” in AS_TIME_SYNC_INIT is set to AS_TS_API, then the time synchronization service in not started until this function is called for the first time.

This function can only be used by the Time Master.

The parameter offset_time specifies the absolute time difference between the local time and the source time in OS_TIMEDATE48 format.

```
typedef struct OS_STR_TIMEDATE48
{
    UINT32 seconds;
    UINT16 ticks;
} OS_TIMEDATE48;
```

The parameter offset_sign specifies the sign of the offset as following:

Value	Meaning
0	local time (offset_time) is higher than source time
1	local time (offset_time) is lower than source time

The parameter time_zone specifies the time difference in minutes with respect to UTC, e.g. -60 for Central European Time.

The parameter daylight specifies the Daylight Saving Time difference in minutes with regards to the standard time. This overrides the daylight saving time offset defined by AS_DAYLIGHT in the Device Configuration.

Returns OK if the device is configured as Time Sync Master and the time source in the Device Configuration is AS_TIME_SYNC_API, otherwise ERROR.

3 CSS EOS API functions

3.1 CORE1 - Core 1 Handler Service

Core1 Services (Core1)

The Core1 service is part of a collection of DSP related services within the EOS.
The Core1 service contains functions for loading Core1 with an executable image and to start the execution of Core1. During the loading of an executable image Core1 will be reset.

---

core1_enable( )

Synopsis

```
void core1_enable(
    UINT32  startAdrs)    /* In: Entry point for Core1 */
```

Description

This routine will bring Core1 out of reset and start the execution on Core1 on provided entry point startAdrs.

---

core1_load_and_run_elf( )

Synopsis

```
INT16 core1_load_and_run_elf(
    char  *pPath)    /* In:  Path, including filename, to elf file */
```

Description

This routine resets Core1 and tries to loads an elf image into memory.
The provided elf image can either be compressed or uncompressed.
First, attempts to load a gzip compressed elf image.
Second, attempts to load an uncompressed elf image.

Upon success, the image is started on Core1.

Returns OK upon success, else ERROR.

---

core1_load_elf( )

Synopsis

```
INT16 core1_load_elf(
    char  *pPath,       /* In:  Path, including filename, to elf file */
    void  **ppEntry)    /* Out: Entry point found in elf file */
```

Description

This routine resets Core1 and tries to loads an elf image into memory.
The provided elf image can either be compressed or uncompressed.
First, attempts to load a gzip compressed elf image.
Second, attempts to load an uncompressed elf image.

Returns OK upon success, else ERROR.

3.2 DSP - DSP Handler Service

DSP Services (DSP)

The DSP services is part of a collection of DSP related services within the EOS.

DSP configurable memory test

It is possible to have max 16 regions of RAM memory areas, which will be CRC supervised.
CSS provides an API for configuring the RAM memory areas and the FPGA will perform the actual supervision with a CRC for each memory area. Memory area 0 - 7 is reserved for applications (available from API), memory area 8 - 14 is reserved for loaded object modules and memory area 15 is reserved for CSS.

The FPGA will supervise each area with a CRC that will be calculated on the first run. In case the FPGA detects a memory corruption it will inform CSS with an interrupt. CSS will then shutdown the device and upon next reboot CSS will write the event to the SE log.

The usage of the DSP memory test are further described in the CSS API's:

```
INT16 dsp_memtest_set(
   UINT32 memorySegmentAreaIndex,
   UINT32 memorySegmentStartAdress,
   UINT32 memorySegmentSize);
```

OCM (On-Chip Memory) parity error check

CSS supports the supervision of OCM parity error checking of the 256kB OCM. The 256kB OCM is devided into four 64kB address ranges.

OCM nr	Size	Address range
OCM0	64 KB	0x00000000 - 0x0000FFFF
OCM1	64 KB	0x00010000 - 0x0001FFFF
OCM2	64 KB	0x00020000 - 0x0002FFFF
OCM3	64 KB	0xFFFF0000 - 0xFFFFFFFF

All four 64kB blocks are supervised with both single and multiple parity error checking enabled.

In case of OCM parity error CSS will only generate a system event error log. But there is also the possibility to add an application callback function, that will be called in case of a parity error, so additional measures can be taken. E.g. a system re-boot.

The application callback function is further described in the CSS API:

```
void dsp_ocm_par_chk_cb_add(VOIDFUNCPTR func);
```

The callback function will be called in a task context with priority 249 and a stack size of 0x4000 bytes.

---

dsp_memtest_get_status( )

Synopsis

```
INT16 dsp_memtest_get_status(
    BOOL    *pTimeout,          /* Out: Memory Supervision timeout */
    UINT32  *pMaxActiveTime)    /* Out: Max scanning time for supervision cycle */
```

Description

Gets status information of the memory supervision.

The available information is the maximum time (in ms) that is used by the memory supervision to scan a supervision cycle and a timeout flag.

By scanning a supervision cycle means that all registered regions are checked.

The timeout flag indicates if the memory supervision is able to check a supervision cycle within 1 second or not. If scanning time exceeds 1 second, the flag is raised.

Both timeout flag and max scanning time are cleared (set to zero) after reading.

Returns:

DSP_TST_OK	upon success
DSP_TST_ERROR	upon error e.g. both inarguments are NULL

dsp_memtest_set( )

Synopsis

```
INT16 dsp_memtest_set(
    UINT32    memorySegmentAreaIndex,   /* In: Memory segment index number      */
                                        /*     Can be any value between 0 - 7   */
    UINT32    memorySegmentStartAdress, /* In: Start address of the memory      */
                                        /*     segment to be supervised.        */
                                        /*     If set to zero (0) the memory    */
                                        /*     supervision of specified         */
                                        /*     memorySegmentAreaIndex will be   */
                                        /*     disabled.                        */
    UINT32    memorySegmentSize)        /* In: Size in bytes of the memory      */
                                        /*     segment to be supervised.        */
                                        /*     If set to zero (0) the memory    */
                                        /*     supervision of specified         */
                                        /*     memorySegmentAreaIndex will be   */
                                        /*     disabled.                        */
```

Description

The function is used for configuring DSP RAM segment supervision.

The FPGA will do the supervision of each RAM segment with a CRC. The CRC will be calculated on the first run and is then continuously supervised by the FPGA. In case the FPGA detects an CRC error on a memory segment an interrupt will be triggerd to CSS. CSS will then force a reboot and upon next reboot the event will be written to the SE log.

An application is able to configure supervison for up to 8 DSP RAM segments, numbered from 0 to 7, which is called the memory segment area index. Each configuration also requires a start adress of the DSP RAM segment and its size in bytes.

In order to disable the supervision of a specific segment the start address and size should be set to zero (0) of the specified segment number index.

Returns:

DSP_TST_OK	upon success
DSP_TST_ERROR	upon error

dsp_ocm_par_chk_cb_add( )

Synopsis

```
STATUS dsp_ocm_par_chk_cb_add(
    VOIDFUNCPTR   fpFunc)             /* In: Function pointer */
```

Description

This function registers a callback function that will be called in case a single or multiple parity error is detected in the OCM (On-Chip Memory).

The callback function will be called in a task context with a task priority of 249 and a stack size of 0x4000 bytes.

The callback function shall be prototyped according to:

```
void ocm_par_chk_cb(UINT32 par_error_type, UINT32 par_error_addr)
```

Parameters:

par_err_type:	The type of parity error. For a single parity error the value will be set to OCM_SINGLE_PARITY_ERROR (0x00000001). For a multiple parity error the value will be set to OCM_MULTIPLE_PARITY_ERROR (0x00000002).
par_error_addr:	The access address associated with the parity error.

To remove the callback function, call the API with a NULL-pointer according to:

```
dsp_ocm_par_chk_cb_add((VOIDFUNCPTR)NULL);
```

Returns OK on success or ERROR in case of failure.

3.3 FPGA - FPGA Handler Service

FPGA Services (FPGA)

The FPGA service provides an API for FPGA issues.

Due to hardware design of EOS board, CSS requires that FPGA code is loaded and is working properly. The FPGA code is closely associated with the hardware revision and it is very important that a correct FPGA code is loaded.

The boot loader will load the flashed FPGA code at booting of target and before CSS is started. However, it is possible to point out a different FPGA code other than the default version. At start-up of CSS, CSS will look for a non-default version on the file system (/app0/) and if it exists, the FPGA code file will be loaded.

There are two formats allowed for the FPGA code on file system and the filenames must match exactly:
- fpga1.bin is an uncompressed version of the FPGA code
- fpga1.bin.gz is a gzip compressed version of the FPGA code

It is not possible to change FPGA code during run-time i.e. after that CSS has been started.

Note, CSS has no opportunity and gives no guarantee that CSS is working correctly when FPGA code is loaded from file system.

---

fpga_get_build_info( )

Synopsis

```
STATUS fpga_get_build_info(
    UINT16     buildInfoLen,      /* In:  Length of pBuildInfo buffer */
    CHAR       *pBuildInfo)       /* Out: FPGA build information */
```

Description

Get FPGA build information

Returns:
OK if type designation could be read
ERROR if input parameter pBuildInfo is NULL

---

fpga_get_revision( )

Synopsis

```
STATUS fpga_get_revision(
    UINT8   *pV,      /* Out: FPGA revision according to v.r.u.b */
    UINT8   *pR,      /* Out: FPGA revision according to v.r.u.b */
    UINT8   *pU,      /* Out: FPGA revision according to v.r.u.b */
    UINT8   *pB)      /* Out: FPGA revision according to v.r.u.b */
```

Description

Get FPGA revision

Returns:
OK if revision could be read
ERROR if any input parameter is NULL

---

fpga_get_revision_core( )

Synopsis

```
STATUS fpga_get_revision_core(
    UINT8   *pV,      /* Out: FPGA Core revision according to v.r.u.b */
```

```
    UINT8   *pR,       /* Out: FPGA Core revision according to v.r.u.b */
    UINT8   *pU,       /* Out: FPGA Core revision according to v.r.u.b */
    UINT8   *pB)       /* Out: FPGA Core revision according to v.r.u.b */
```

Description

Get FPGA Core revision

Returns:

OK	if revision could be read
ERROR	if any input parameter is NULL

fpga_get_type_designation( )

Synopsis

```
STATUS fpga_get_type_designation(
    UINT16  typeDesignationLen,    /* In:  Length of p_type_designation buffer */
    CHAR    *pTypeDesignation)     /* Out: FPGA type designation */
```

Description

Get FPGA type designation

Returns:

OK	if type designation could be read
ERROR	if input parameter p_type_designation is NULL

fpga_is_fpga1_loaded( )

Synopsis

```
BOOL fpga_is_fpga1_loaded(
    void)
```

Description

Get information about if FPGA code is loaded or not

Returns:

TRUE	if FPGA code is loaded
FALSE	if no FPGA code is loaded

fpga_show( )

Synopsis

```
void fpga_show(
    void)
```

Description

Display information of loaded FPGA code e.g. type designation, revision and build information.

4 Processor Interface Library

4.1 CSS PIL Service

CSS implements the Processor Interface Library. This document includes a brief description of the PIL API.

---

pil_accept_queue( )

Synopsis

```
char* pil_accept_queue(
    short sID,                  /* IN : Message queue identification */
    short* psErr)               /* OUT: Detailed error info */
```

Description

This routine retrieves a pointer to a message in a message queue specified by sID.

If there is a message, its pointer is returned, otherwise an error code is returned in psErr.

The task is never suspended, if no message is available the routine returns immediately.

A returned message is implicitly removed from queue.

This routine may not be called from an ISR.

Returns the following value with the status code in psErr (if not NULL):

Return value	psErr	Description
Message pointer	PI_RET_OK	if a message in the queue was available
NULL	PI_ERR_ID	if sID is out of range or invalid
NULL	PI_ERR_NM	if no message was available

---

pil_alloc( )

Synopsis

```
void* pil_alloc(
    UINT16 size)                /* IN : Size of memory block */
```

Description

This routine allocates size bytes of memory and returns a pointer to the beginning of the allocated block.

Returns NULL if there is not enough contiguous memory available in the heap.

---

pil_call_hw_int( )

Synopsis

```
void pil_call_hw_int(
    short sNumber)              /* IN: Interrupt number with respect to the
                                       interrupt vector table */
```

Description

This routine simulates the hardware interrupt with number sNumber by calling interrupt handler routine.

Not supported for PowerPC platforms.

---

pil_copy16( )

Synopsis

```
void pil_copy16(
    void* pTarget,              /* IN : Pointer to target address */
    void* pSource,              /* IN : Pointer to source address */
    unsigned short Size)        /* IN : Number of words to copy */
```

Description

This routine copies the first Size words from pSource to pTarget one word at a time.

This may be desirable if a buffer can only be accessed with word instructions, as in certain word-wide memory-mapped peripherals. The source and destination must be word-aligned.

pil_copy8( )

Synopsis

```
void pil_copy8(
    void* pTarget,              /* OUT: Pointer to target address */
    void* pSource,              /* IN : Pointer to source address */
    unsigned short Size)        /* IN : Number of bytes to copy */
```

Description

This routine copies the first Size bytes from pSource to pTarget one byte at a time.

This may be desirable if a buffer can only be accessed with byte instructions, as in certain byte-wide memory-mapped peripherals.

pil_create_queue( )

Synopsis

```
short pil_create_queue(
    short sSize,                /* IN : Number of entries in the queue */
    short sOption,              /* IN : Strategy PI_ORD_PRIO Priority
                                   PI_ORD_FCFS FIFO */
    short* psErr)               /* OUT: Detailed error info:
                                   PI_RET_OK - Success
                                   PI_ERR_MM - No memory - Internal
                                   sem. ID destroyed */
```

Description

This routine creates a new message queue with sSize message entries.

Two possible strategies can be specified with sOption:

PI_ORD_PRIO	priority-serve
PI_ORD_FCFS	first-come-first-serve.

This routine may not be called from an ISR.

Returns an identifier for the created message queue on success or ERROR if the creation failed with the status code PI_ERR_MM stored in psErr (if not NULL).

pil_create_semaphore( )

Synopsis

```
short pil_create_semaphore(
    unsigned short sInit,       /* IN : Initial semaphore value */
    short sOption,              /* IN : PI_ORD_PRIO    Priority
                                        PI_ORD_FCFS    First Come First Serve */
    short* psErr)               /* OUT: Detailed error info */
```

Description

This routine creates a counting semaphore with the specified initial counting value.

An initial value of zero in the parameter sInit will initially block the semaphore.

Handling of tasks attempting to access a blocked semaphore can be determined by the parameter sOption:

Option	Description
PI_ORD_PRIO	the task with the highest priority is the
-	first to get the unblocked semaphore
PI_ORD_FCFS	the first task which has taken the semaphore is the
-	first to get the unblocked semaphore

If the semaphore creation was successful PI_RET_OK is stored in the parameter psErr, otherwise PI_ERR_NC

The call can affect the task scheduling if a higher priority task is waiting for a semaphore taken by a task with lower priority.

This function may not be called from an ISR.

Returns the created semaphore identification number or ERROR on failure.

pil_create_timeout( )

Synopsis

```
short pil_create_timeout(
    short* psID,                /* OUT: Timeout identification */
    void* pFunc,                /* IN : Pointer to called function */
    void* pPara,                /* IN : Pointer to parameters of function */
    short sCount)               /* IN : Number of timeouts if PI_FOREVER is specified
                                        the function is called infinitely many times */
```

Description

This routine defines a timeout call to a function, which will be executed after setting the timeout value with pil_enable_timeout().

The identifier for the timeout is stored in the parameter psID.

The parameter pFunc is the pointer to the function for the timeout call.

The parameter pPara is the pointer to the arguments in the call to the function for the timeout call.

The parameter sCount determines, how many times given function is called if PI_FOREVER is specified the function is called infinitely many times.

This function may not be called from an ISR.

Returns PI_RET_OK on success or PI_ERR_MM if no timeout is available.

pil_delete_timeout( )

Synopsis

```
short pil_delete_timeout(
    short sID)                  /* IN: Timeout identification */
```

Description

This routine deletes a timeout mechanism specified by sID.

Before calling this routine, the timeout mechanism must be created and disabled resp. the timeout has to be elapsed.

After deleting a timeout, the identifier for the timeout is invalid and no further calls to it are allowed.

This function may not be called from an ISR.

Returns PI_RET_OK on success, otherwise PI_ERR_ID if sID is out of range or invalid.

pil_disable( )

Synopsis

```
void pil_disable(
    void)
```

Description

This routine disables the global interrupt, i.e. interrupts are not serviced until pi_enable is called.

pil_disable_timeout( )

Synopsis

```
short pil_disable_timeout(
    short sID)                  /* IN: Timeout identification */
```

Description

This routine disables an enabled timeout specified by sID.

Before calling this function, the timeout must be created and enabled. The function pil_enable_timeout() may re-enable the timeout.

This function may not be called from an ISR.

Returns the following value:

Return value	Description
PI_RET_OK	on success
PI_ERR_ID	if sID is out of range or invalid
PI_ERR_ID	if the timeout can not be canceled

pil_enable( )

Synopsis

```
void pil_enable(
    void)
```

Description

This routine enables the global interrupt.

The routine pil_disable must have been called before this routine.

pil_enable_timeout( )

Synopsis

```
INT16 pil_enable_timeout(
    INT16 sID,                  /* IN: Timeout identification */
    UINT32 ulTicks)             /* IN: Ticks until timeout if PI_FOREVER
                                       timeout never occurs */
```

Description

This routine enables a timeout mechanism specified by sID.

After the number of ticks specified in the parameter ulTicks has elapsed, the function specified in the call to pil_create_timeout() is called.

This function may not be called from an ISR.

Returns the following value:

Return value	Description
PI_RET_OK	on success
PI_ERR_ID	if sID is out of range or invalid
PI_ERR_CT	this timeout is already enabled

pil_free( )

Synopsis

```
void pil_free(
    void* ptr)                  /* IN : Pointer to Memory Block */
```

Description

This routine returns a previously allocated memory block specified by ptr.

pil_inquiry_semaphore( )

Synopsis

```
unsigned short pil_inquiry_semaphore(
    short sID,                  /* IN : Semaphore identification */
    short* psErr)               /* OUT: Detailed error info */
```

Description

This routine makes an inquiry on a counting semaphore specified by sID.

To be sure the returned counter value is valid, encapsulate this routine by calls of pil_disable() and pil_enable().

In the parameter psErr the status of the inquiry is stored (if psErr is not NULL). The status is PI_RET_OK on success or PI_ERR_ID if sID does not exist.

Returns the current semaphore value if the parameter psErr (if not NULL) is PI_RET_OK.

pil_lock_task( )

Synopsis

```
void pil_lock_task(
    void)
```

Description

This routine prevents rescheduling. No other task (not even one with higher priority) will be allowed to run.

This routine may not be called from an ISR.

pil_memset16( )

Synopsis

```
void pil_memset16(
    void* targetp,              /* IN: Pointer to memory to be modified */
    short value,                /* IN: 16 bit word to write */
    unsigned short size)        /* IN: Number of words to be written */
```

Description

This function sets a block of memory pointed out by targetp with size number of 16 bit words to what is specified in value.

Can be used when a faster algorithm than pil_memset8 is wanted.

pil_memset8( )

Synopsis

```
void pil_memset8(
    void* targetp,              /* IN: Pointer to memory to be modified */
    char value,                 /* IN: Byte to write */
    unsigned short size)        /* IN: Number of bytes to be written */
```

Description

This routine sets a block of memory pointed ot by targetp with size number of bytes to what is specified in value.

Can be used for memory chips allowing only one byte write access.

pil_pend_semaphore( )

Synopsis

```
void pil_pend_semaphore(
    INT16 sID,                  /* IN : Semaphore identification number */
    INT32 lTimeout,             /* IN : After timeout has expired, the task is
                                        reactivated unconditionally.
                                        If PI_FOREVER is specified, task is only
                                        reactivated, if the semaphore becomes available. */
    INT16* psErr)               /* OUT: Detailed error info */
```

Description

This routine makes the calling task pending on a counting semaphore specified by sID.

The semaphore counter is decremented each time this function is called. If the semaphore is not available the task is suspended until a timeout occurs or until semaphore becomes available.

If more than one task is waiting for the semaphore to be unblocked, then the strategy applies in order to decide which task will be restarted, see pil_create_semaphore().

The parameter lTimeout is the timeout in ticks to wait for the semaphore to be available.

A status code is stored in the parameter psErr (if not NULL) as follows:

Status code	Description
PI_RET_OK	on success
PI_ERR_TO	if the timeout has expired
PI_ERR_TO	if lTimeout is 0 (zero) but sID is unavailable
PI_ERR_ID	if sID is out of range or invalid

This function may not be called from an ISR.

pil_post_semaphore( )

Synopsis

```
void pil_post_semaphore(
    short sID,                  /* IN : Semaphore identification */
    short* psErr)               /* OUT: Detailed error info */
```

Description

This routine makes a counting semaphore specified by sID available.

The semaphore counter is incremented each time this function is called and PI_RET_OK is stored in the parameter psErr.

When the semaphore value reaches maximum counter value, an overflow error PI_ERR_OV is stored in the parameter psErr.

If sID is out of range or invalid PI_ERR_ID is stored in the parameter psErr (if not NULL).

The call can affect the task scheduling if higher priority task is waiting for posting semaphore.

This function may not be called from an ISR.

pil_queue_inquiry( )

Synopsis

```
void pil_queue_inquiry(
    short sID,                  /* IN : queue-ID */
    short* countp,              /* OUT: > 0  messages available
                                        = 0  no message
                                        = -1 sID is invalid */
    short* psErr)               /* OUT: Detailed error info */
```

Description

This routine evaluates the number of messages waiting a message queue specified by sID.

The number of messages waiting is stored in the parameter countp.

A status code is stored in the parameter psErr (if not NULL) as follows:

Status code	Description
PI_RET_OK	on success
PI_ERR_MM	if countP is NULL
PI_ERR_ID	if sID is out of range or invalid

pil_queue_jam( )

Synopsis

```
void pil_queue_jam(
    short sID,                  /* IN : Don't use VxWorks queue-ID, only PIL one. */
    const char* msg,            /* IN : Message to send. Empty string
                                        (and even NULL) is allowed */
    short* psErr)               /* OUT: Detailed error info */
```

Description

This routine writes a pointer to a message at the first position in a message queue specified by sID.

Only the pointer to the message is sent to the queue and not the message itself so the pointer must be a pointer to a const messsage.

The parameter msg is a pointer to a const char buffer.

The call can affect the task scheduling if a higher priority task is waiting for a message in the same message queue.

A status code is stored in the parameter psErr (if not NULL) as follows:

Status code	Description
PI_RET_OK	on success
PI_ERR_ID	if sID is out of range or invalid or if msg is NULL
PI_ERR_FB	if the message queue is full

pil_receive_queue( )

Synopsis

```
char* pil_receive_queue(
    INT16 sID,                  /* IN : Message queue identification */
    INT32 lTimeout,             /* IN : After timeout has expired, the task is
                                        reactivated unconditionally. If PI_FOREVER
                                        is specified, task is only restarted,
                                        if message is sent to the queue.*/
    INT16* psErr)               /* OUT: Detailed error info */
```

Description

This routine retrieves a pointer to a message in a message queue specified by sID.

The task is suspended until a timeout occurs or until a message is placed into the message queue by another task. If there is at least one message, the appropriate message is returned and removed from the message queue and the task changes to ready state.

The parameter lTimeout specifies a timout to wait for a message in ticks. If PI_FOREVER is specified the task is waiting for an infinite of time for a message to arrive.

This routine may not be called from a ISR.

Returns the following value with the status code in psErr (if not NULL):

Return value	psErr	Description
Message pointer	PI_RET_OK	if a message in the queue was available
NULL	PI_ERR_ID	if sID is out of range or invalid
NULL	PI_ERR_TO	if the timout has expired

pil_semaphore_accept( )

Synopsis

```
void pil_semaphore_accept(
    short sID,                  /* IN : Semaphore identifier */
    short* psErr)               /* OUT: Detailed error info */
```

Description

This routine makes it possible for the calling task to take a counting semaphore without the risk of beeing blocked.

Instead of beeing blocked, the routine returns immediately and PI_ERR_PD is stored in the parameter errp.

The semaphore counter is decremented each time this function is called and PI_RET_OK is stored in the parameter errp.

If sID is out of range or invalid PI_ERR_ID is stored in the parameter psErr (if not NULL).

The call can affect the task scheduling if higher priority task is waiting for posting semaphore.

This function may not be called from an ISR.

---

pil_send_queue( )

Synopsis

```
void pil_send_queue(
    short sID,                  /* IN : Message queue identification */
    const char* pchMsg,         /* IN : Message to send. Empty string
                                        (and NULL) is allowed */
    short* psErr)               /* OUT: Detailed error info */
```

Description

This routine writes pointer to a message in a message queue specified by sID.

Only the pointer to the message is sent to the queue and not the message itself so the pointer must be a pointer to a const messsage.

The parameter pchMsg is a pointer to a const char buffer.

The call can affect the task scheduling if a higher priority task is waiting for a message in the same message queue.

A status code is stored in the parameter psErr (if not NULL) as follows:

Status code	Description
PI_RET_OK	on success
PI_ERR_ID	if sID is out of range or invalid
PI_ERR_FB	if the message queue is full

---

pil_task_delay( )

Synopsis

```
void pil_task_delay(
    UINT32 ticks)                /* IN: Delay time in ticks */
```

Description

This routine delays the calling tasks for the number of ticks specified in time.

This routine may not be called from an ISR.

---

pil_unlock_task( )

Synopsis

```
void pil_unlock_task(
    void)
```

Description

This routine allows rescheduling again.

This routine may not be called from an ISR.

5 Revision History

Revision	Date	Document state	Review Record
__	2016-01-22	Released	3EST000232-5335
_A	2016-07-06	Released	–
_B	2016-09-23	Released	–
_C	2017-03-30	Released	3EST000232-5335
_D	2018-12-12	Released	3EST000232-5335
_E	2019-05-29	Released	3EST000232-5335
_F	2020-01-22	Released	3EST000232-5335

6 Active Sheet Record

Rev Level	Section	Description of change

Index