Technical Document

GENERIC PROTOCOL STACK FRAMEWORK

GPF

VSI/PEI – FRAME/BODY INTERFACES

FUNCTIONAL INTERFACE DESCRIPTION

Document Number:	06-03-10-ISP-0002
Version:	0.17
Status:	Draft
Approval Authority:	
Creation Date:	1999-May-05
Last changed:	2006-Apr-06 by MP
File Name:	vsipei_api.doc
ECCN:	US: 5D991 Europe: EAR99

Important Notice

Change History

Date	Changed by	Approved by	Version	Status	Notes
1999-May-05	MP et al.		0.1		1
1999-May-18	MS et al.		0.2		2
1999-Jul-15	MP et al.		0.3		3
1999-Nov-01	HJS et al.		0.4		
1999-Dec-07	MP et al.		0.5		4
2000-Jan-21	HJS et al.		0.6		5
2000-Oct-11	MP		0.7		6
2001-Sep-07	MP		0.8		7

2001-Dec-01	MP		0.9		8
2003-May-20	XINTEGRA		0.10	Draft	
2003-Sep-08	MP		0.11		9
2003-Dec-08	MP		0.12		10
2005-Feb-08	MP		0.13		11
2005-Apr-13	MP		0.14		12
2005-May-11	MP		0.15		13
2005-Nov-24	MP		0.16		14
2005-Dec-09	MP		0.17		15
2006-Mar-24	MP		0.18		16

Notes:

1. Initial version
2. English/Format check
3. Review
4. Reference C_8415.036 added
5. New Template (MS)
6. Type T_VOID_STRUCT added
7. Vsi_c/m_status added
8. Dynamic allocation functions added
9. vsi_e_handle(), vsi_e_name, vsi_c_status() added
10. general update
11. added non-blocking memory allocation API
12. added PSEND_TO_PC macro, minor corrections
13. added FMALLOC and FPALLOC_SDU
14. added trace registration for non-GSP entities
15. added virtual memory pool API
16. added realloc

Table of Contents

List of Figures and Tables

List of References

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

06-03-10-UDO-0001	frame_users_guide.doc	FUG – Frame Users Guide
06-03-10-ISP-0003	os_api.doc	OS - Operating System Interface
06-03-42-UDO-0001	str2ind_userguide.doc	Compressed/Binary Trace

1.1 Abbreviations

VSI	Virtual System Interface
PEI	Protocol Stack Entity Interface
RTOS	Real-time Operating System

2 Introduction

G23 is a software package implementing Layers 2 and 3 of the ETSI-defined GSM air interface signaling protocol, and as such represents the part of a GSM protocol software which is both, platform and manufacturer independent. Therefore, G23 can be viewed as a building block providing standardized functionality through generic interfaces for easy integration.

The G23 suite of products consists of the following items:

- Layers 2 and 3 for speech & short message services,
- Layers 2 and 3 for fax & data services,
- Application Control Interface,
- Slim MMI [02.30] and
- Test and integration support tools.

This document is the Functional Interface Description for the interfaces between the body and the frame - the Virtual System Interface (VSI) and the Protocol Entity Interface (PEI).

3 Frame/Body Concept

The frame body concept has been designed in the context of the G23 Protocol Stack. In the G23 Protocol Stack, a process represents the protocol logic of a protocol stack entity. This architecture separates the process functionality into two logical modules, the process frame and the process body. Common process functionality is located in the process frame. The main process functionality is located in the process body.

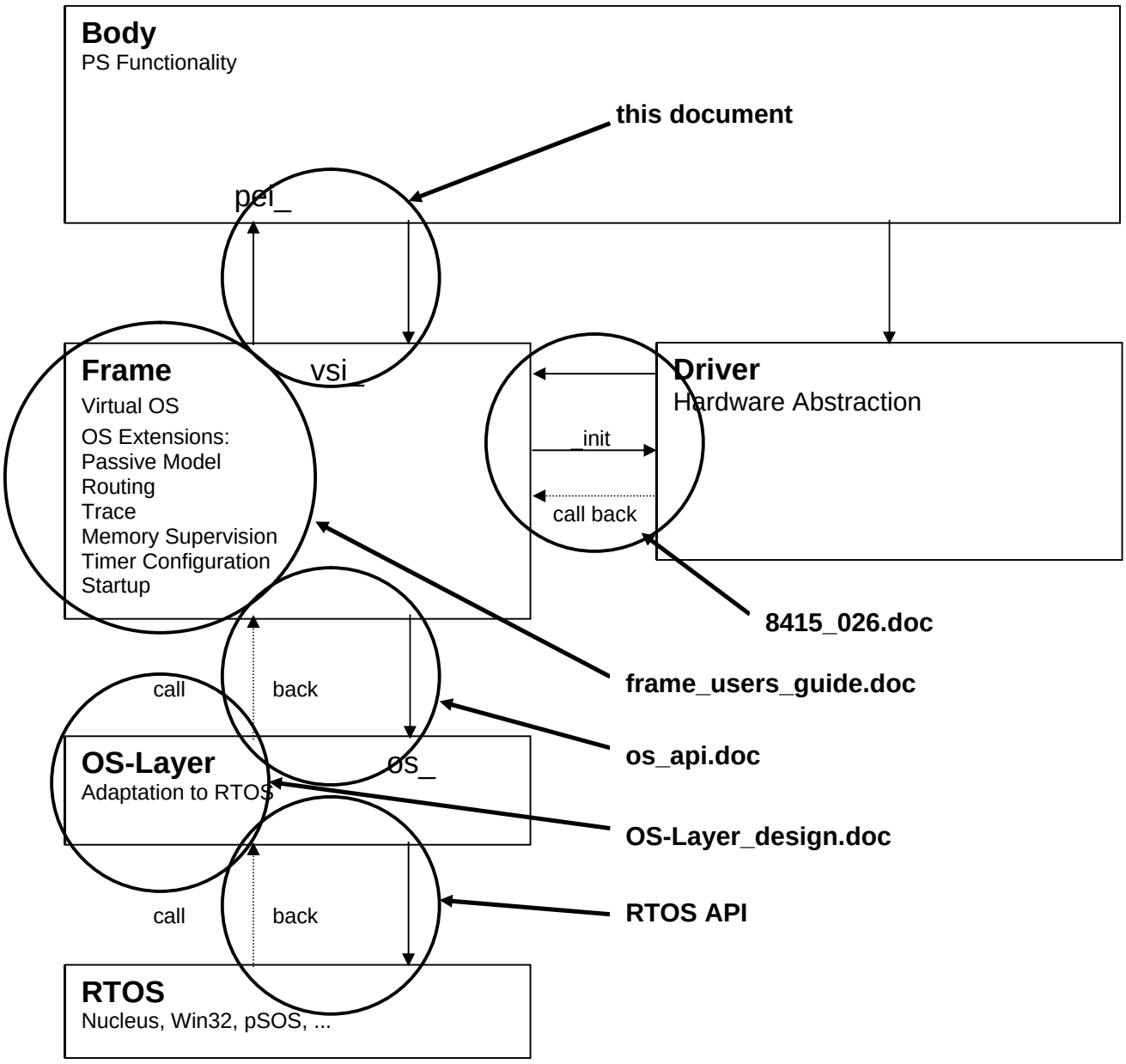

Figure 1: Protocol Stack Software Architecture and Documentation

As Figure 1 shows, the interfaces between the body and the frame are the Protocol Entity Interface (PEI) and the Virtual System Interface (VSI).

The Protocol Entity Interface (PEI) is the functional interface of the protocol entities to be accessed by the frame.

A protocol stack entity is registered to the system by calling the pei_create() function that exports the user defined entity creation parameters.

The communication between the protocol stack entities is done via message queues. These message queues are created by the frame during the system startup and the communication between the entities is opened by the function pei_init(). A message identifier defines the type of the received queue entry. Three different kinds of entries can be transferred through these queues: primitives, signals and timeouts. Each of these message types has a corresponding PEI function that is called if a message has been received.

Setting of dynamic configurations of protocol stack entities e.g. timer configuration as well as reading of these configurations is performed by the function pei_config().

The Virtual System Interface (VSI) is the functional interface of the frame seen by the body. For the bodies, the VSI serves as an interface to the underlying OS Layer that is an abstraction of the RTOS (Nucleus, Win32,...). The RTOS resources created are accessed through the VSI that calls OS Layer functions, some of them e.g. the timers are managed in the VSI. The VSI also provides access to extended frame functionality such as tracing and memory supervision.

The frame/body concept requires that access to the bodies is performed only via the PEI and that the bodies must use the VSI to access the RTOS or the frame.

3.1 Variants and Options

3.1.1 Active/Passive Body

A protocol stack entity can run in two different variants. In the active variant, the main loop is located within the entity in the function pei_run() where the corresponding input message queue has to be supervised. If a primitive has been received, the function pei_primitive(), pei_signal() or pei_timeout() is called for further evaluation. In the passive variant, the main loop is located outside the body in the frame. The procedures for this variant are the same as in the active case if a message has been received, except the PEI functions are called by the frame.

3.1.2 Communication

There are two different methods of communication between the protocol stack entities. If the communication is done by reference every time a primitive has to be sent, a partition from the partition memory pool is allocated and filled with the primitive data. The address of this partition is written into the input queue of the destination entity. This entity frees the partition after processing its contents. If the communication is carried out in this manner, a buffer must never be used by the source entity after it has been sent with vsi_c_psend() as the partition may have already been freed by the receiver and used by other processes.

If the communication is done by copy the contents of a buffer filled by a entity are copied into a buffer provided by the destination entity.

4 Interfaces

4.1 Data Types

4.1.1 Base Types

The following type names are used as synonyms for hardware/compiler dependent integer types:

SHORT	16 bit, signed
USHORT	16 bit, unsigned
LONG	32 bit, signed
ULONG	32 bit, unsigned

4.1.2 T_HANDLE

Definition: This is an integral type, therefor the definition is not given here. The size may depend from the underlying RTOS.

Description: This type is used for all parameters containing VSI handles (for tasks, queues, timers, semaphores).

4.1.3 T_TIME

Definition: This is an integral type, therefor the definition is not given here. The size may depend from the underlying RTOS but it is at least 31 bits.

Description: This type is used for all parameters containing a time value (which is always in msec).

4.1.4 T_PRIM_HEADER

Definition:

```
typedef struct
{
  ULONG opc;          /* operation code of primitive                 */
  ULONG len;          /* primitive length                            */
  ULONG use_cnt;      /* counter indicates current number of users   */
  T_sdu *sdu;         /* pointer to sdu struct if available          */
  ULONG dph_offset;   /* offset of dynamic prim header               */
  ULONG sh_offset;    /* offset of system header                     */
} T_PRIM_HEADER
```

Description: Some of the elements are used by the PEI or VSI. The parameter opc is the operation code of a primitive, which is needed to execute an opc-dependent body function from a table located in the PEI Interface. Len is the length of the primitive.

4.1.5 T_DP_HEADER

Definition:
```
typedef struct
{
    ULONG magic_nr;      /* magic number is checked at each access */
    ULONG size;          /* available bytes in dynamic primitive partition */
    ULONG use_cnt;       /* counter indicates current number of users */
    ULONG offset;        /* offset from partition begin to next free byte */
    T_VOID_STRUCT**
      drp_bound_list;
                  /* pointer to the list of dynamic partitions bound to this partition */
    struct _T_DP_HEADER *next;
} T_DP_HEADER
```

Description: This header is used to handle chains of dynamic partitions. It can either resist at the end of a communication primitive or in front of an independent root partition. It is important that its size matches exactly the size of T_PRIM_HEADER and that the use_cnt field is at the same position in both headers. Given this, e.g, a generic free-function can be used for primitive and dynamic root pointers.

4.1.6 T_VOID_STRUCT

Definition: typedef unsigned long T_VOID_STRUCT

Description: Pointers of type T_VOID_STRUCT are passed to functions in order to avoid warnings when using void pointers with a subsequent cast operation within the called function.

4.1.7 T_PEI_FUNC

Definition:
```
typedef struct
{
  SHORT (*pei_init)(T_HANDLE);
  SHORT (*pei_exit)(void);
  SHORT (*pei_primitive)(void*);
  SHORT (*pei_timeout)(USHORT);
  SHORT (*pei_signal)(ULONG,void*);
  SHORT (*pei_run)(T_HANDLE,T_HANDLE);
  SHORT (*pei_config)(char*);
  SHORT (*pei_monitor)(void**);
} T_PEI_FUNC
```

Description: The structure of the type T_PEI_FUNC contains the addresses of the protocol entities pei functions.

4.1.8 T_PEI_INFO

Definition:
```
typedef struct
{
  const char *      name;
  T_PEI_FUNC        pei_table;
  ULONG             stackSize;
  USHORT            queueEntries;
  USHORT            priority;
  USHORT            num_of_timers,
  USHORT            flags;
} T_PEI_INFO
```

Description: The structure of the type T_PEI_INFO contains the information to create an entity. The following table contains a list of the data elements and corresponding descriptions.

name	name of the entity
pei_table	table with addresses of the bodies PEI functions
stackSize	maximum stacksize of the entity
queueEntries	number of entries in the input queue
priority	priority of the entity (0=low, 255=high)
num_of_timers	number of timers needed for this entity
flags	ACTIVE_BODY(Bit 0): active(0)/passive(1) variant of body COPY_BY_REF(Bit 1): communication by copying buffers(0)/only references are passed through the queues(1) SYSTEM_PROCESS (Bit 2): frame internal use TRC_NO_SUSPEND (Bit 3): discard traces if no memory available or TST queue full PARTITION_AUTO_FREE (Bit 4): automatic partition deallocation when returning from pei_primitive PRIM_NO_SUSPEND (Bit 5): discard routed primitives if no memory available or TST queue full INT_DATA_TASK (Bit 6): allocate task stack and queue memory from internal RAM pool ADD_QUEUE_ENTRIES (Bit7): add queue sizes for grouped entities (default is to take biggest value)

4.1.9 T_VSI_VMP_FLOW_CTRL_PARAM

Definition:

```
typedef struct
{
  U32   off_level;
  U32   on_level;
} T_VSI_VMP_FLOW_CTRL_PARAM
```

Description: The structure of the type T_VSI_VMP_CONFIG_PARAM contains the parameters for the flow control. Both levels are the amount in bytes when flow control changes between on and off.

off_level	this is the lower level where flow control is switched off
on_level	this is the upper level where flow control is switched on

4.1.10 T_VSI_VMP_NOTIFY_PARAM

Definition:
```
typedef struct
{
  T_HANDLE   entity_handle;
  void       (*signal_status)(U32 vmp_handle, S32 flow_ctrl_state);
} T_VSI_VMP_NOTIFY_PARAM
```

Description: The structure of the type T_VSI_VMP_CONFIG_PARAM contains the parameters for the flow control. Both levels are the amount in bytes when flow control changes between on and off.

entity_handle	entity handle to be used for asynchronous notification of flow control state changes
on_level	user callback function to be called for synchronous notification of flow control state changes

4.1.11 T_VSI_VMP_CONFIG_PARAM

Definition:
```
typedef struct
{
  U32                    size_vmp;
  T_VSI_FLOW_CTRL_PARAM  flow_ctrl;
  U32                    flags;
} T_VSI_VMP_CONFIG_PARAM
```

Description: The structure of the type T_VSI_VMP_CONFIG_PARAM contains the properties of a virtual memory pool created with vsi_vmp_create(). The amount of memory available in a virtual pool is specified by its *size*. It is passed to vsi_vmp_create(), see 4.5.5.1).

size_vmp	size of virtual memory pool in bytes		
flow_ctrl	parameters for flow control		
flags	bit 0	VSI_FAST_MEMORY	memory pool in fast memory
	bit 1	VSI_MEM_NON_BLOCKING	Non-blocking allocation
	bit 2..31		reserved for future use, need to be zero

4.1.12 T_ VMP_GUARD

Definition:
```
typedef U32 T_VMP_GUARD
```

Description: This type is used for a special guard pattern at the behind the virtual pool header in a memory block.

4.1.13 T_QMSG

Definition:

```
struct
{
        USHORT              MsgType;
        union
        {
                struct T_Prim
                {
                        T_VOID_STRUCT   *Prim;
                        ULONG           PrimLen;
                };
                struct Signal
                {
                        ULONG           SigOPC;
                        T_VOID_STRUCT   *SigBuffer;
                        ULONG           SigLen;
                };
                struct Timeout
                {
                        ULONG           Index;
                };
        }
} T_QMSG;
```

Description: The structure of the type T_QMSG defines the message transported via message queues. MsgType indicates the kind of the message (MSG_..., see 4.2.2).

4.2 Constants

4.2.1 Return Codes

PEI_OK	0	successful execution
PEI_ERROR	-1	error
VSI_OK	0	successful execution
VSI_TIMEOUT	1	no success during specified time
VSI_VMP_NOT_CLEAN	2	there are still allocations from that virtual pool
VSI_REF_CNT_NZ	3	the memory block is still referenced and not freed
VSI_ERROR	-1	error
VSI_INVALID_PARAM	-2	a given parameter was invalid
VSI_NO_MEMORY	-3	there is no memory available from that virtual pool

4.2.2 Message Types

MSG_PRIMITVE	1	indicates a primitive
MSG_SIGNAL	2	indicates a signal
MSG_TIMEOUT	3	indicates a timeout

4.2.3 Trace Classes

TC_FUNC	0x00000001	trace class for function names	TRACE_FUNCTION()
TC_EVENT	0x00000002	trace class for events	TRACE_EVENT()
TC_PRIM	0x00000004	trace class for primitives	PTRACE_IN
TC_STATE	0x00000008	trace class for states	GET_STATE/SET_STATE()
TC_SYSTEM	0x00000010	trace class for frame info	frame internal
TC_ISIG	0x00000020	trace class for internal signals	TRACE_ISIG()
TC_ERROR	0x00000040	trace class for errors	TRACE_ERROR()
TC_CCD	0x00000080	trace class for CCD	CCD internal
TC_TIMER	0x00000100	trace class for timers	frame internal
TC_DATA	0x00000200	trace class for data	TRACE_MEMORY/PRIMITIVE()
TC_SDU	0x00000400	trace class for SDUs	TRACE_SDU()
TC_PROFILER	0x00000800	trace class for profiler	frame internal
TC_USER1	0x00010000	trace class for users	TRACE_USER_CLASS()
TC_USER2	0x00020000	trace class for users	TRACE_USER_CLASS()
TC_USER3	0x00040000	trace class for users	TRACE_USER_CLASS()
TC_USER4	0x00080000	trace class for users	TRACE_USER_CLASS()
TC_USER5	0x00100000	trace class for users	TRACE_USER_CLASS()
TC_USER6	0x00200000	trace class for users	TRACE_USER_CLASS()
TC_USER7	0x00400000	trace class for users	TRACE_USER_CLASS()
TC_USER8	0x00800000	trace class for users	TRACE_USER_CLASS()

4.2.4 Timer Configuration

TIMER_SET	1
TIMER_RESET	2
TIMER_SPEED_UP	3
TIMER_SLOW_DOWN	4
TIMER_SUPPRESS	5
TIMER_CLEAN	6

4.2.5 Object identifier

OS_OBJSYS	to get system information (see vsi_i_object())
OS_OBJTASK	to get task information (see vsi_i_object())
OS_OBJQUEUE	to get queue information (see vsi_i_object())
OS_OBJPARTITIONGROUP	to get partition group information (see vsi_i_object())
OS_OBJMEMORYPOOL	to get memory pool information (see vsi_i_object())
OS_OBJTIMER	to get timer information (see vsi_i_object())
OS_OBJSEMAPHORE	to get semaphore information (see vsi_i_object())

These constants are defined in the OS interface description os_api.doc (06-03-10-ISP-0003).

4.2.6 Other constants

OS_NOTASK special task handle to indicate that a call is performed from outside any task ('non-task thread', e.g. interrupt routine) (defined in os_api.doc (06-03-10-ISP-0003).

4.3 Macros for to call VSI Functions

The following macros should be used for allocation, deallocation and reusing of a partition of the primitive partition memory pool and for sending primitives and signals. The usage of these macros simplifies the communication because the user does not have to care about things like primitive header initialization or the calling of memory supervision functions.

All memory allocation macros are available in two versions. The one with underscore like PALLOC create a pointer and initialize it, the macros with underscore like P_ALLOC do not create but return a pointer. All other macros are also available in an 'underscore variant' to keep the naming consistent.

The macros with the extension _NB are the non-blocking variant of the memory allocation macros. These return a NULL pointer in no memory is available.

It is strictly recommended to use these macros. The attempt to manage without these macros may result in severe problems if any of the feature normally done by the macros is missed.

4.3.1 PALLOC, PALLOC_NB

Syntax: PALLOC (*PrimitiveVariable, PrimitiveName*)
Purpose: Allocate memory for a primitive of the type indicated by *PrimitiveName* and store the memory address in *PrimitiveVariable*.
Description: A variable named *PrimitiveVariable* of type T_*PrimitiveName** is defined.
Memory is allocated by calling the function vsi_c_pnew() (see 4.5.3.10). The size of the allocated memory chunk is sizeof(T_PRIM_HEADER) + sizeof(T_*PrimitiveName*) bytes.
The address of the allocated memory is stored in the defined variable *PrimitiveVariable*.
Initialization: The operation code is set to *PrimitiveName* in the header of the primitive (opc).
The length is set to sizeof(T_PRIM_HEADER)+sizeof(T_*PrimitiveName*) in the header of the primitive(len).
The 'sdu' is initialized in the header of the primitive with the value 0.
Example: PALLOC (rr_data_req, RR_DATA_REQ);
rr_data_req->sapi = 3;

4.3.2 PALLOC_GENERIC

Syntax: PALLOC_GENERIC (*PrimitiveName, PartitionPoolGroup, flags*)
Purpose: Macro allows to allocate from a different partition pool group than the default one for primitives. The parameter *PartitionPoolGroup* needs to be filled with the group handle from the PoolGroupHandle array in xxxcomp.c, refer to the frame_users_guide.doc.
The parameter flags allows to specify the blocking behavior. If VSI_MEM_NON_BLOCKING is set, a NULL pointer will be returned when no memory is available, otherwise the allocation function will block.

4.3.3 P_ALLOC, P_ALLOC_NB

Syntax: *Pointer* = P_ALLOC_NB (*PrimitiveName*)
Purpose: Macro does the same as PALLOC but returns the memory address instead of creating a pointer and assign the address to it.

4.3.4 PALLOC_DESCx, PALLOC_DESCx_NB

Syntax: PALLOC_DESCx (*PrimitiveVariable, PrimitiveName*)
Purpose: Does the same as PALLOC but additionally the element dph_offset in the primitive header is set to the offset of the element T_desclistx in the allocated primitive. This allows the frame to scan the descriptor list of a primitive when checking its integrity or tracking partition ownership.

4.3.5 PALLOC_MSG, PALLOC_SDU

Syntax: PALLOC_MSG (*PrimitiveVariable, PrimitiveName, MessageName*)
PALLOC_SDU (*PrimitiveVariable, PrimitiveName, MessageSizeInBits*)
FPALLOC_SDU (*PrimitiveVariable, PrimitiveName, MessageSizeInBits*)
Purpose: Allocate memory for one primitive of the type indicated by *PrimitiveName*. The Primitive contains an SDU big enough to carry a message
- of the type indicated by *MessageName* (PALLOC_MSG) or
- which contains *MessageSizeInBits* bits (PALLOC_SDU).

The memory address is stored in *PrimitiveVariable.*
Description: A variable named *PrimitiveVariable* of type T_*PrimitiveName** is defined.
Memory is allocated by calling the function vsi_c_new_sdu() (see 4.5.3.14). The size of the allocated memory chunk is
sizeof(T_PRIM_HEADER) + sizeof(T_*PrimitiveName*) bytes +
BSIZE_*MessageName* (PALLOC_MSG)
sizeof(T_PRIM_HEADER) + sizeof(T_*PrimitiveName*) bytes + *MessageSizeInBits* (PALLOC_SDU)
bits.
The address of the allocated memory is stored in the defined variable *PrimitiveVariable*.
The <*MessageSizeInBits*> must be the name of a C-variable or a literal constant or named constant. The <*MessageSizeInBits*> cannot be a delibrate C-expression. This limitation stems from a simple mechanism introduced to avoid a mismatch of PALLOC_MSG and PALLOC_SDU. This mechanism will lead to a compile time error for an expression like PALLOC_SDU (DL_DATA_REQ) if DL_DATA_REQ is a name of a message (meant to be used in PALLOC_MSG).
Initialization: The operation code is set to *PrimitiveName* in the header of the primitive.
The length is set to sizeof(T_PRIM_HEADER)+sizeof(T_*PrimitiveName*) in the header of the primitive(len). Note that the message size is not contained in 'len'.
The 'sdu' is initialized in the header of the primitive.
The components 'l_buf' and 'o_buf' of the SDU are initialized.
Example: PALLOC_MSG (rr_data_req, RR_DATA_REQ, U_HANDOV_FAIL);
rr_data_req->sapi = 3;

4.3.6 FPALLOC_SDU

Purpose: FPALLOC_SDU has the same functionality as PALLOC_SDU but allocate from the fast memory pool is the option FF_FAST_MEMORY is set. If not set FPALLOC_SDU is defined to PALLOC_SDU.

4.3.7 P_ALLOC_SDU, FP_ALLOC_SDU, P_ALLOC_MSG

Syntax: *Pointer* = P_ALLOC_SDU (*PrimitiveName, MessageName*)
Pointer = FP_ALLOC_SDU (*PrimitiveName, MessageName*)
Pointer = P_ALLOC_MSG (*PrimitiveName, MessageSizeInBits*)

Purpose: Macro do the same as PALLOC_SDU and PALLOC_MSG but return the memory address instead of creating a pointer and assign the address to it.

4.3.8 DRPO_ALLOC

Syntax: *Pointer* = DRPO_ALLOC (*PrimitiveName, Guess*)
Purpose: Allocate memory for the root of a dynamic sized primitive of the type indicated by *PrimitiveName* and return the pointer to it.
Description: Memory is allocated by calling the function vsi_drpo_new() (see 4.5.4.7). If the caller has an idea of the total size of the dynamic sized primitive he can use the parameter *Guess* to reserve the needed space in the allocated partition. If the totally needed memory is unknown then *Guess* can be set to zero. The size of the allocated memory chunk is sizeof(T_PRIM_HEADER) + sizeof(T_D_HEADER) + sizeof(T_*PrimitiveName*) + Space with Space = *Guess* if *Guess* != 0 resp. Space = sizeof(T_*PrimitiveName*)*3 if *Guess* == 0.
The address of the allocated memory is returned.
Initialization: The operation code in the primitive is set to *PrimitiveName*.
Example:

```
msg = DRPO_ALLOC(CPHY_CONFIG_REQ,0);
msg->config_id = config_id;
```

4.3.9 DRP_ALLOC

Syntax: *Pointer* = DRP_ALLOC (*Size, Guess*)
Purpose: Allocate memory for the root of a chunk of dynamic memory and return the pointer to it.
Description: Memory is allocated by calling the function vsi_drp_new() (see 4.5.4.8). If the caller has an idea of the total amount of needed memory he can use the parameter *Guess* to reserve the needed space in the allocated partition. If the totally needed memory is unknown then *Guess* can be set to zero. The size of the allocated memory chunk is sizeof(T_D_HEADER) + *Size* + Space with Space = *Guess* if *Guess* != 0 resp. Space = sizeof(T_*PrimitiveName*)*3 if *Guess* == 0.
The address of the allocated memory is returned.
Example:

```
size = sizeof(T_RRC_RB_RB_CONF);
guess = UMTS_AS_ASN1_MAX_RB_MUX_OPTIONS *
sizeof(T_UMTS_AS_ASN1_rb_mapping_option);
ptr = (T_RRC_RB_RB_CONF *)DRP_ALLOC(size,guess);
```

4.3.10 DRP_BIND

Syntax: DRP_BIND (*Child, Parent*)
Purpose: Bind a child root-pointer (or a primitive-pointer) to a parent root-pointer.
Description: The parent has to allocated with DRPO_ALLOC or DRP_ALLOC. The child has to be allocated with DRPO_ALLOC, DRP_ALLOC or one of the PALLOC-like macros. DRP_BIND adds child to the internal drp_bound_list of parent and (recursively) increases the child use_cnt. When FREE is called for parent FREE is also called for all bound childs.
Example:

```
parent=DRPO_ALLOC(...);
child=DRP_ALLOC(...);
if (DRP_BIND(child,parent)==VSI_OK)
{
        parent->ptr_elem=child;
}
else
{
        parent->ptr_elem=NULL;
}
FREE(child);
/* the actual child memory will be freed together with parent */
```

4.3.11 DP_ALLOC

Syntax: *Pointer* = DP_ALLOC (*Size, addr, Guess*)
Purpose: Allocate additional memory in the chain specified by *addr* and return the pointer to it.
Description: Memory is allocated by calling the function vsi_dp_new() (see 4.5.4.8). If the caller has an idea of the total amount of needed memory he can use the parameter *Guess* to reserve the needed space in the allocated partition. If the totally needed memory is unknown then *Guess* can be set to zero. The size of the allocated memory chunk is sizeof(T_D_HEADER) + *Size*) + Space with Space = *Guess* if *Guess* != 0 resp. Space = sizeof(T_*PrimitiveName*)*3 if *Guess* == 0.
The address of the allocated memory is returned.
Example:
```
size = sizeof (T_UMTS_AS_ASN1_UL_DCCH_MSG_MSG);
guess = sizeof(T_UMTS_AS_ASN1_active_set_update_failure);
msg_ptr = (T_UMTS_AS_ASN1_UL_DCCH_MSG_MSG*) DRP_ALLOC (size,
guess);
msg_ptr->msg_type = UMTS_AS_ASN1_MSG_UL_DCCH_MSG;
msg_ptr->msg_data->msg->ptr_body = (T_ul_dcch_msg_type__body*)
DP_ALLOC(size,msg_ptr,0);
```

4.3.12 PREUSE, P_REUSE

Syntax: PREUSE (*PrimitiveVariable0, PrimitiveVariable, PrimitiveName*)
P_REUSE has the same syntax.
Purpose: Re-use a primitive loosing the contents of the primitive.
Description: An existing primitive *PrimitiveVariable0* is re-used. The contents of the primitive is lost. The operation code is changed to the value supplied by *PrimitiveName*. The length is set to sizeof(T_PRIM_HEADER) + sizeof(T_*PrimitiveName*) in the header of the primitive(len). The SDU offset is initialized in the header of the primitive with the value 0. The new Variable *PrimitiveVariable* is defined and initialized with the value of *PrimitiveVariable0*. The variable *PrimitiveVariable0* should not be used after PREUSE. The function vsi_ppm_reuse() (see 4.5.9.4) is called for memory supervision.
Example:
```
PREUSE (rr_establish_req, rr_release_ind, RR_RELEASE_IND);
```

4.3.13 PREUSE_MSG, PREUSE_SDU, P_REUSE_MSG, P_REUSE_SDU

Syntax: PREUSE_MSG (*PrimitiveVariable0, PrimitiveVariable, PrimitiveName, MessageName*)
PREUSE_SDU (*PrimitiveVariable0, PrimitiveVariable, PrimitiveName, MessageSizeInBits*)
P_REUSE_MSG and P_REUSE_SDU have the same syntax.
Purpose: Re-use a primitive loosing the contents of the primitive.
Entry state: PS_RECEIVED
Exit state: PS_ALLOCATED
Description: An existing primitive *PrimitiveVariable0* is re-used. The contents of the primitive is lost. The operation code is changed to the value supplied by *PrimitiveName*. The length is set to sizeof(T_PRIM_HEADER) + sizeof(T_*PrimitiveName*) in the header of the primitive(len). Note that the message size is not contained in 'len'. The SDU offset is initialized in the header of the primitive. The components 'l_buf' and 'o_buf' of the SDU are initialized. The new Variable *PrimitiveVariable* is defined and initialized with the value of *PrimitiveVariable0*. The variable *PrimitiveVariable0* should not be used after PREUSE. The notification procedure is called.
The <*MessageSizeInBits*> must be the name of a C-variable or a literal constant or named constant. The <*MessageSizeInBits*> cannot be a delibrate C-expression. This limitation stems from a simple mechanism introduced to avoid a mismatch of PREUSE_MSG and PREUSE_SDU. This mechanism will lead to a compile time error for an expression like PREUSE_SDU (DL_DATA_REQ) if DL_DATA_REQ is a name of a message (meant to be used in PREUSE_MSG).

4.3.14 PATTACH, P_ATTACH

Syntax: PATTACH (*primitive*)
Purpose: Attach to the primitive determined by *primitive*.
Description: The reference counter in the primitive header incremented.
PATTACH calls the function vsi_c_pattach() (see 4.5.4.6).
ATTENTION: PATTACH can only attach to primitives allocated with DRPO_ALLOC, DRP_ALLOC, PALLOC, PALLOC_SDU, PALLOC_MSG and PALLOC_DESCx.
Example: PATTACH (primitive);

4.3.15 PFREE, P_FREE

Syntax: PFREE (*PrimitiveVariable*)
Purpose: Release the memory previously allocated with PALLOC or PALLOC_MSG.
Description: The contents of *PrimitiveVariable* is the address of the memory to be freed.
The memory de-allocation function vsi_c_free() (see 4.5.3.14) is called.
Example: PFREE (rr_data_req);

4.3.16 MALLOC, MALLOC_NB

Syntax: MALLOC (*Variable, Size*)
Purpose: Allocate memory of *Size* Bytes from the communication partition pool and store the memory address in *Variable*.
Description: A variable named *Variable* of a pointer type must exist.
Memory is allocated by calling the function vsi_m_cnew() (see 4.5.4.2).
The address of the allocated memory is stored in the variable *Variable*.
Example: MALLOC (buffer, BUFFERSIZE);

4.3.17 FMALLOC

Syntax: FMALLOC (*Variable, Size*)
Purpose: If the option FF_FAST_MEMORY is set FMALLOC allocates a buffer of *Size* Bytes from the fast partition pool and store the memory address in *Variable*. If not set FMALLOC is defined to MALLOC.
Description: A variable named *Variable* of a pointer type must exist.
Memory is allocated by calling the function vsi_m_cnew() (see 4.5.4.2).
The address of the allocated memory is stored in the variable *Variable*.
Example: FMALLOC (buffer, BUFFERSIZE);

4.3.18 MREALLOC, MREALLOC_NB

Syntax: MREALLOC (*ptr, ptr0, Size*)
Purpose: Provide sufficient space in the memory block specified by *ptr0* to store *Size* bytes in total. This macro allocates from the primitive partition pool.
Description: A variable named *ptr* of a pointer type must exist.
If the allocated memory block specified by *ptr0* is sufficiently large to store *Size* bytes, ptr0 is returned. If the allocated block specified by *ptr0* is too small, a memory block big enough to store *Size* bytes is allocated, the contents of *ptr0* is copied to the newly allocated block and the address of the allocated memory block is stored in the variable *ptr*. Memory is allocated by calling the function vsi_m_cnew() (see 4.5.4.2). If *ptr0* is NULL a memory block of *Size* bytes is allocated. If *Size* is equal to zero and *ptr0* is not NULL, the memory block specified by *ptr0* is freed.
Example: MREALLOC (big_block, small_block, BLOCKSIZE);

4.3.19 MALLOC_GENERIC

Syntax: MALLOC_GENERIC (*Variable , PartitionPoolGroup, flags*)
Purpose: Macro allows to allocate from a different partition pool group than the default one for primitives. The parameter *PartitionPoolGroup* needs to be filled with the group handle from the PoolGroupHandle array in xxxcomp.c, refer to the frame_users_guide.doc. The parameter flags allows to specify the blocking behavior. If VSI_MEM_NON_BLOCKING is set, a NULL pointer will be returned when no memory is available, otherwise the allocation function will block.

4.3.20 MALLOC_DESCx, MALLOC_DESCx_NB

Syntax: MALLOC_DESCx (*Variable, Size*)
Purpose: Does the same as MALLOC but additionally the descriptor type is set in the element desc_type of the memory header. This allows the frame to follow the descriptor list elements during integrity check or ownership tracking.

4.3.21 M_ALLOC, M_ALLOC_NB, FM_ALLOC

Syntax: *Pointer* = M_ALLOC (*Size*)
Purpose: Macro does the same as MALLOC but returns the memory address instead of assigning the address to the passed pointer.

4.3.22 M_REALLOC, M_REALLOC_NB

Syntax: *Pointer* = M_REALLOC (*ptr*, *Size*)
Purpose: Macro does the same as MREALLOC but returns the memory address instead of assigning the address to the passed pointer.

4.3.23 M_ALLOC_DESCx, M_ALLOC_DESCx_NB

Syntax: *Pointer* = M_ALLOC_DESCx (*Size*)
Purpose: Macro does the same as MALLOC_DESCx but returns the memory address instead of assigning the address to the passed pointer.

4.3.24 DMALLOC, DMALLOC_NB

Syntax: DMALLOC (*Variable, Size*)
Purpose: Allocate memory of *Size* Bytes from the non-communication partition pool and store the memory address in *Variable*.
Description: A variable named *Variable* of a pointer type must exist.
Memory is allocated by calling the function vsi_m_new() (see 4.5.4.1).
The address of the allocated memory is stored in the variable *Variable*.
Example: DMALLOC (buffer, BUFFERSIZE);

4.3.25 DMREALLOC, DMREALLOC_NB

Syntax: DMREALLOC (*ptr, ptr0, Size*)
Purpose: Provide sufficient space in the memory block specified by *ptr0* to store *Size* bytes in total. This macro allocates from the general purpose partition pool DMEM.
Description: A variable named *ptr* of a pointer type must exist.
If the allocated memory block specified by *ptr0* is sufficiently large to store *Size* bytes, ptr0 is returned. If the allocated block specified by *ptr0* is too small, a memory block big enough to store *Size* bytes is allocated, the contents of *ptr0* is copied to the newly allocated block and the address of the allocated memory block is stored in the variable *ptr*. Memory is allocated by calling the function vsi_m_cnew() (see 4.5.4.2). If *ptr0* is NULL a memory block of *Size* bytes is allocated. If *Size* is equal to zero and *ptr0* is not NULL, the memory block specified by *ptr0* is freed.
Example: DMREALLOC (big_block, small_block, BLOCKSIZE);

4.3.26 D_ALLOC, D_ALLOC_NB

Syntax: *Pointer* = D_ALLOC (*Size*)
Purpose: Macro does the same as MALLOC but returns the memory address instead of assigning the address to the passed pointer.

4.3.27 D_REALLOC, D_REALLOC_NB

Syntax: *Pointer* = D_REALLOC (*ptr*, *Size*)
Purpose: Macro does the same as DMREALLOC but returns the memory address instead of assigning the address to the passed pointer.

4.3.28 MATTACH, M_ATTACH

Syntax: MATTACH (*memory*)
Purpose: Attach to the memory partition determined by *memory*.
Description: The reference counter in the memory partition header is incremented.
MATTACH calls the function vsi_m_attach() (see 4.5.4.6).
ATTENTION: MATTACH can only attach to memory allocated with MALLOC, M_ALLOC and MALLOC_DESCx.
Example: MATTACH (memory);

4.3.29 FREE

Syntax: FREE (*Variable*)
Purpose: Release the memory previously allocated with any of the allocation macros DRPO_ALLOC, DRP_ALLOC, PALLOC, PALLOC_SDU, PALLOC_MSG and PALLOC_DESCx
Description: The contents of *Variable* is the address of the memory to be freed.
The memory de-allocation function vsi_free() (see 4.5.4.12) is called.
Example: FREE (buffer);

4.3.30 MFREE, M_FREE

Syntax: MFREE (*Variable*)
Purpose: Release the memory previously allocated with MALLOC and MALLOC_DESCx
Description: The contents of *Variable* is the address of the memory to be freed.
The memory de-allocation function vsi_m_free() (see 4.5.4.4) is called.
Example: MFREE (buffer);

4.3.31 DMFREE, D_FREE

Syntax: DMFREE (*Variable*)
Purpose: Release the memory previously allocated with DMALLOC.
Description: The contents of *Variable* is the address of the memory to be freed.
The memory de-allocation function vsi_m_free() (see 4.5.4.4) is called.
Example: DMFREE (buffer);

4.3.32 PSEND, P_SEND

Syntax: PSEND (*Receiver*, *PrimitiveVariable*)
P_SEND has the same syntax.
Purpose: Send a primitive identified by *PrimitiveVariable* to the *Receiver*.
Description: The function vsi_c_psend() (see 4.5.3.5) which transmits the primitive is called.
Example: PSEND (RR, rr_data_req);
Note: The primitive referenced by *PrimitiveVariable* may or may not have an SDU. This is detected by the 'sdu' component in the primitive header.

4.3.33 PSEND_CALLER, P_SEND_CALLER

Syntax: PSEND (*Caller, Receiver*, *PrimitiveVariable*)
P_SEND has the same syntax.
Purpose: Send a primitive identified by *PrimitiveVariable* to the *Receiver*. Compared to PSEND/P_SEND, the handle of the calling entity is passed to the macro and not calculated by the frame. This macro can be use to send from outside the GPF context and pretend to be a GPF entity, e.g. in callback functions.
Description: The function vsi_c_psend_caller() (see 4.5.3.6) which transmits the primitive is called.
Example: PSEND_CALLER (RR, rr_data_req);
Note: The primitive referenced by *PrimitiveVariable* may or may not have an SDU. This is detected by the 'sdu' component in the primitive header.

4.3.34 PRIM_SEND_TO_PC

Syntax: PRIM_SEND_TO_PC (*Caller, Receiver*, *PrimitiveId*)
Purpose: Send a allocated primitive identified by *PrimitiveId to a* GPF based application named *Receiver* on the connected PC in a primitive.
Description: The function vsi_o_primsend() (see 4.5.8.9) which sends the primitive to the PC is called.
Example:
PRIM_SEND_TOPC(*cc_handle,"PCO",prim*); sends *prim* to PCO

4.3.35 DATA_SEND_TO_PC

Syntax: DATA_SEND_TO_PC (*Caller, Filter, Receiver*, *PrimitiveId, Pointer, Length*)
Purpose: Send any other data identified by *Pointer* and *Length* to the *Receiver* on the connected PC in a primitive.
Description: If the trace filter class *Filter* is enabled the frame allocates a memory partition for a primitive, copies the data identified by *Pointer* and *Length* into this primitive and sends it with the id *PrimitiveId* to a GPF based application named *Receiver* on the connected PC. The function vsi_o_primsend() (see 4.5.8.9) is called.
Example: DATA_SEND_TO_PC(*cc_handle,TC_PRIM,"PCO",PRIM_ID,string,strlen(string)*); sends the *string* in a primitive with id *PRIM_ID* to PCO

4.3.36 PSIGNAL, P_SIGNAL

Syntax: PSIGNAL (*Receiver*, *PrimitiveName, PrimitiveVariable*)
P_SIGNAL has the same syntax.
Purpose: Send a primitive as a signal.
Description: The primitive data (without a header) is send as a signal (i.e. with higher priority than a normal primitive) to the *Receiver*. No memory is allocated. The *PrimitiveName* is used as the signal opcode. It has to be a symbolic constant and the type T_*PrimitiveName* must exist. The function vsi_c_send() (see 4.5.3.4) which transmits the signal is called.
Example: PSIGNAL (ACI, RA_DATA_IND, ra_data->tra.ra_data_ind);

4.3.37 PACCESS, P_ACCESS

Syntax: PACCESS (*PrimitiveVariable*)
P_ACCESS has the same syntax.
Purpose: Update file and line of primitive access in the partition status if MEMORY_SUPERVISION active.
Description: The function vsi_ppm_access() (see 4.5.9.5) is called to monitor the access of a primitive.
Example: PACCESS (rr_activate_req);

4.3.38 PPASS, P_PASS

Syntax: PPASS (*OldPrimitiveVariable, NewPrimitiveVariable,PrimitiveName*)
P_PASS has the same syntax as PPASS.
Purpose: PPASS creates a new variable *NewPrimitiveVariable* initializes it with *OldPrimitiveVariable* and sets the opc in the header of *NewPrimitiveVariable* to the opc from *OldPrimitiveVariable*. It calls the VSI function vsi_c_ppass() (4.5.3.16).
Example: PPASS(cci_decomp_ind, sn_data_ind, SN_DATA_IND);

4.3.39 TRACE_FUNCTION, TRACE_FUNCTION_P1…9

Syntax: TRACE_FUNCTION (*FunctionName*)
Purpose: TRACE_FUNCTION is used to trace the names of entered functions and calls the trace API function vsi_o_func_ttrace() resp. vsi_o_func_itrace() if compressed tracing is used. Function traces can be en/disabled via the bit TC_FUNC, refer to 4.2.3. TRACE_FUNCTION_P1…9 can be used trace a format string plus a set of arguments.
Example: TRACE_FUNCTION (“pei_primitive”);
TRACE_FUNCTION_P1(“pei_primitive (%x)”,prim);

4.3.40 TRACE_EVENT, TRACE_EVENT_P1…9

Syntax: TRACE_EVENT (*EventString*)
Purpose: TRACE_EVENT is used to trace any information and calls the trace API function vsi_o_event_ttrace() resp. vsi_o_event_itrace() if compressed tracing is used. Event traces can be en/disabled via the bit TC_EVENT, refer to 4.2.3. TRACE_EVENT_P1…9 can be used trace a format string plus a set of arguments.
Example: TRACE_EVENT (“start coding of …”);
TRACE_EVENT_P2(“Value1 = %d, Value2 = %d”, val1, val2);

4.3.41 TRACE_USER_CLASS, TRACE_USER_CLASS_P1…9

Syntax: TRACE_USER_CLASS (*TraceString*)
Purpose: TRACE_USER_CLASS is used to trace any information with a user defined trace class and calls the trace API function vsi_o_class_ttrace() resp. vsi_o_class_itrace() if compressed tracing is used. User traces can be en/disabled via the bits TC_USER1…8, refer to 4.2.3. TRACE_USER_CLASS_P1…9 can be used trace a format string plus a set of arguments.
Example: TRACE_USER_CLASS (TC_USER1, “start coding of…”);
TRACE_USER_CLASS_P2(TC_USER5“Value1 = %d, Value2 = %d”, val1, val2);

4.3.42 PTRACE_IN, PTRACE_OUT

Syntax: PTRACE_IN (*opc*), PTRACE_OUT (*opc*)
Purpose: PTRACE_IN/OUT is used to trace the names of received/sent primitives and calls the trace API function vsi_o_ptrace(). Primitive traces can be en/disabled via the bit TC_PRIM, refer to 4.2.3. PTRACE_OUT() is called in vsi_c_psend() when calling PSEND (4.3.32) and does not need to be called by any entity.
Example: PTRACE_IN (*RR_ACTIVATE_REQ*);

4.3.43 TRACE_ERROR

Syntax: TRACE_ERROR (*error_string*)
Purpose: TRACE_ERROR is used to trace error conditions and calls the trace API function vsi_o_error_ttrace() resp. vsi_o_error_itrace() if compressed tracing is used. Error traces can be en/disabled via the bit TC_ERROR, refer to 4.2.3. TRACE_ERROR must not be confused with TRACE_ASSERT() (4.3.44) to detect and handle fatal errors which force a reset of the mobile.
Example: TRACE_ERROR (“Invalid … received”);

4.3.44 TRACE_ASSERT

Syntax: TRACE_ASSERT (*expression*)
Purpose: TRACE_ASSERT is used to detect fatal error conditions and calls the trace API function vsi_o_assert() if the expression passed to TRACE_ASSERT() is false. TRACE_ASSERT cannot be disabled via any TC_… mask. TRACE_ASSERT must not be confused with TRACE_ERROR() (4.3.43) to trace error conditions.
Example: TRACE_ASSERT (value > 0);

4.3.45 TRACE_MEMORY

Syntax: TRACE_MEMORY (*source, pointer, length*)
Purpose: TRACE_MEMORY is used to trace (as text string) any data specified by *pointer* and *length* and calls the trace API function vsi_o_memtrace(). The parameter *source* determines the sender entity displayed in PCO and is the entity handle passed to the entities pei_init() function. Memory traces can be en/disabled via the bit TC_DATA, refer to 4.2.3.
Example: TRACE_MEMORY (rr_handle, rr_data_base, sizeof(rr_data_base));

4.3.46 TRACE_HEXDUMP

Syntax: TRACE_HEXDUMP (*source, pointer, length*)
Purpose: TRACE_HEXDUMP is used to trace (as special primitive) any data specified by *pointer* and *length* and calls the trace API function vsi_o_primsend(). The parameter *source* determines the sender entity displayed in PCO and is the entity handle passed to the entities pei_init() function. Memory traces can be en/disabled via the bit TC_DATA, refer to 4.2.3.
Example: TRACE_HEXDUMP (rr_handle, rr_data_base, sizeof(rr_data_base));

4.3.47 TRACE_MEMORY_PRIM

Syntax: TRACE_MEMORY_PRIM (*source, destination, opc, pointer, length*)
Purpose: TRACE_MEMORY_PRIM is used to send any data specified by *pointer* and *length* as a primitive with the opcode *opc* to PCOand calls the trace API function vsi_o_primsend(). The parameter *source* determines the destination entity displayed in PCO and is the entity handle passed to the entities pei_init() function. The parameter *destination* determines the original receiver of the primitive displayed by PCO. These traces can be en/disabled via the bit TC_DATA, refer to 4.2.3.
TRACE_MEMORY_PRIM is intended to be used to trace the data passed via function call interfaces between two entities. The primitive may be specified in an SAP document to be decoded in PCO.
Example: TRACE_MEMORY_PRIM (l1_handle, grr_handle, MAC_DATA_IND, ptr, length);

4.3.48 TRACE_USER_CLASS_MEMORY_PRIM

Syntax: TRACE_USER_CLASS_MEMORY_PRIM (*source, traceclass, destination, opc, pointer, length*)
Purpose: TRACE_USER_CLASS_MEMORY_PRIM is used to send any data specified by *pointer* and *length* as a primitive with the opcode *opc* to PCO and calls the trace API function vsi_o_primsend(). The parameter *source* determines the destination entity displayed in PCO and is the entity handle passed to the entities pei_init() function. Via parameter *traceclass* the trace class can be specified which needs to be enabled for the tracing entity to switch on this macro (refer to 4.2.3). The parameter *destination* determines the original receiver of the primitive displayed by PCO.
TRACE_USER_CLASS_MEMORY_PRIM is intended to be used to trace the data passed via function call interfaces between two entities. The primitive may be specified in an SAP document to be decoded in PCO.
Example: TRACE_USER_CLASS_MEMORY_PRIM (l1_handle, TC_USER1, grr_handle, MAC_DATA_IND, ptr, length);

4.3.49 TRACE_SDU

Syntax: TRACE_SDU (*source, destination, entity, direction, type, pointer, length*)
Purpose: TRACE_SDU is used to send an encoded air message (sdu) specified by *pointer* and *length* as a primitive with the opcode SDU_TRACE_OPC to PCO and calls the trace API function vsi_o_sdusend(). The parameter *source* determines the sender and the parameter *destination* the receiver entity displayed in PCO. Via the parameters *entity, direction* and *type* PCO will get the information needed to decode the sdu:
- *entity*: number of message catalogue, e.g., retrievable via ccddata_get_ccdent(<entity-name, e.g., “RR”>) ... use TRCSDU_NO_ENTITY if the sdu contains a PD/TI byte by which the entity can be obtained automatically
- *direction*: direction of air message, TRCSDU_DIR_UPLINK or TRCSDU_DIR_DOWNLINK
- *type*: message type number ... use TRCSDU_NO_MSGTYPE if the sdu contains a msg-id byte by which the type number can be obtained automatically

Example: TRACE_SDU (l1_handle, grr_handle, ccddata_get_ccdent(“RR”),
TRCSDU_DIR_UPLINK, 0x12, ptr, length);

4.3.50 TRACE_IP

Syntax: TRACE_IP (*source, destination, direction, pointer, length*)
Purpose: TRACE_IP is used to send an IP package specified by *pointer* and *length* as a primitive with the opcode IP_TRACE_OPC to PCO and calls the trace API function vsi_o_primsend(). The parameter *source* determines the sender and the parameter *destination* the receiver entity displayed in PCO. The *direction* parameter can be UPLINK_OPC or DOWNLINK_OPC.
Example: TRACE_IP (l1_handle, grr_handle, DOWNLINK_OPC, ptr, length);

4.3.51 PRF_LOG_FUNC_ENTRY

Syntax: PRF_LOG_FUNC_ENTRY (*funcname*)
Purpose: PRF_LOG_FUNC_ENTRY is used to inform the profiler on entering a function and calls the profiler API function prf_log_func_entry(). Profiler API calls can be en/disabled via the bit TC_PROFILER, refer to 4.2.3.
Example: PRF_LOG_FUNC_ENTRY (“pei_primitive”);

4.3.52 PRF_LOG_FUNC_EXIT

Syntax: PRF_LOG_FUNC_EXIT (*funcname*)
Purpose: PRF_LOG_FUNC_EXIT is used to inform the profiler on leaving a function and calls the profiler API function prf_log_func_exit(). Profiler API calls can be en/disabled via the bit TC_PROFILER, refer to 4.2.3.
Example: PRF_LOG_FUNC_EXIT (“pei_primitive”);

4.3.53 PRF_LOG_POI

Syntax: PRF_LOG_POI (*point of interest*)
Purpose: PRF_LOG_POI is used to inform the profiler on reaching a point of interest and calls the profiler API function prf_log_point of interrest(). Profiler API calls can be en/disabled via the bit TC_PROFILER, refer to 4.2.3.
Example: PRF_LOG_POI (“decoding of … started”);

4.4 Protocol Entity Interface (PEI)

4.4.1 pei_create () - Create the Protocol Stack Entity

Function definition:

SHORT pei_create (T_PEI_INFO ** info)

Parameters:

Type	Name	Meaning	
T_PEI_INFO **	info	entity setup data	OUT

Return:

Type	Meaning	
SHORT	PEI_OK	Success

Options: none

Description:

The function pei_create () exports the startup configuration data of a protocol stack entity. This function is used to register this entity in the system so that the frame can allocate the required system resources such as an input queue or a number of timers, know the addresses of the other PEI functions and can start the task with specified priority and stacksize.

This function must be called before any other pei_*() function.

4.4.2 pei_init () - Initialize Protocol Stack Entity

Function definition:

SHORT pei_init (T_HANDLE handle)

Parameters:

Type	Name	Meaning	
T_HANDLE	handle	handle of the entity	IN

Return:

Type	Meaning	
SHORT	PEI_OK	Success
	PEI_ERROR	Error

Options: none

Description:

In the passive variant, the function pei_init () is called by the frame at initialization. This function should be called by the function pei_run() in the active body configuration (see 2.1.1). This function stores the entity handle that is needed for vsi calls of this entity.

The function pei_init() opens the communication with other entities by calling the function vsi_c_open() and storing the returned communication handles. If a communication resource cannot be found the pei_init () function returns an error.

The function pei_init() is also responsible for body-specific initializations.

4.4.3 pei_exit () - Close Resources and Terminate

Function definition:

SHORT pei_exit (void)

Parameters: ---------

Return:

Type	Meaning	
SHORT	PEI_OK	Success
	PEI_ERROR	Error

Options: none

Description:

The function pei_exit () releases all resources reserved by the entity so that they are available for other purposes. It closes the communication resources with other entities and eventually frees allocated memory.

In the active variant, this function sets a flag that enables the pei_run() function to exit its main loop and terminate the entity.

4.4.4 pei_primitive () - Process Primitive

Function definition:

SHORT pei_primitive (void * primitive)

Parameters:

Type	Name	Meaning	
void *	primitive	primitive buffer	IN

Return:

Type	Meaning	
SHORT	PEI_OK	Primitive processed
	PEI_ERROR	Primitive not processed

Options: None

Description:

In the passive variant, this function pei_primitive() is called by the frame if a primitive for this entity has been received. Primitive processing is done by evaluating the opc in the header and calling a specific function assigned to the opc.

In the active variant, this function should be called by pei_run ()(see 2.1.1).

If the communication is carried out by exchanging pointer references (see 2.1.2) it must be ensured that the protocol stack entity has discarded the primitive buffer or has reused the buffer and sent the reference to another entity.

4.4.5 pei_signal () - Process Signal

Function definition:

SHORT pei_signal (ULONG opc, void * data)

Parameters:

Type	Name	Meaning	
ULONG	opc	operation code of signal	IN
void *	data	pointer to signal data	IN

Return:

Type	Meaning	
SHORT	PEI_OK	Success
	PEI_ERROR	Error

Options: None

Description:

In the passive variant, the function pei_signal() is called by the frame if a signal is received. This function will never be called while a pei_primitive() call is active in the same body.

In the active variant, this function should be called by pei_run() if a signal is received.

4.4.6 pei_timeout () - Process Timeout

Function definition:

SHORT pei_timeout (USHORT index)

Parameters:

Type	Name	Meaning	
USHORT	index	index of expired timer	IN

Return:

Type	Meaning	
SHORT	PEI_OK	Success
	PEI_ERROR	Error

Options: None

Description:

In the passive variant, the function pei_signal() is called by the frame if a timeout of a previous started timer occurred. The parameter index is the same that was transferred to the VSI when the timer was started.

In the active variant, this function should be called by pei_run() if a timeout occurred.

This function will never be called while a pei_primitive() call is active in the same body.

4.4.7 pei_run () - Process Primitive

Function definition:

SHORT pei_run (T_HANDLE taskhandle, T_HANDLE comhandle)

Parameters:

Type	Name	Meaning	
T_HANDLE	taskhandle	task handle	IN
T_HANDLE	comhandle	handle of own queue	IN

Return:

Type	Meaning	
SHORT	PEI_OK	Success
	PEI_ERROR	Error

Options: only active Variant

Description:

The function pei_run() is used if the main loop of the finite state machine, i.e. waiting for primitives, is located in the protocol stack (active variant of protocol stack, see 2.1.1).

Waiting for primitives can be implemented by using the function vsi_c_await ().

The function pei_run () should return if the environment has invoked the function pei_exit ().

4.4.8 pei_config () - Dynamic Configuration

Function definition:

SHORT pei_config (char * inString)

Parameters:

Type	Name	Meaning	
char *	inString	configuration string	IN

Return:

Type	Meaning	
SHORT	PEI_OK	Success
	PEI_ERROR	Error

Options: None

Description:

The function pei_config () is used to set configuration values of a protocol entity.

The configuration string inString is evaluated. If it is a command to set configuration data, this is done by direct access to the protocol stack entity data structures or (for timers) by calling the function vsi_t_config() that modifies the timer configuration tables.

The format of configuration commands is defined in [C_8410.003].

4.4.9 pei_monitor () - Monitoring of Physical Parameters

Function definition:

SHORT pei_monitor (void ** monitor)

Parameters:

Type	Name	Meaning	
void **	monitor	address of monitor struct	IN

Return:

Type	Meaning	
SHORT	PEI_OK	Success
	PEI_ERROR	Error

Options: None

Description:

With the function pei_monitor (), the environment requests the address of the monitor struct in the protocol entity. The monitor struct includes relevant physical parameters of the protocol entity. The parameters are updated cyclically. This way the environment has always the possibility of accessing parameters of the protocol stack. These parameters are used to create monitor reports that are transferred to a display or test system in order to generate statistical data outside the functionality of a protocol stack but with access to protocol stack parameters.

It is acceptable to read the parameters of the monitor struct, but it is absolutely unacceptable to write to the monitor struct. The content of the monitor struct is custom specific.

One parameter of the monitor struct is a version number of the protocol stack entity.

4.5 Virtual System Interface (VSI)

The function describing the VSI can be divided into different sections depending on the functionality:

- tasks
- communication
- memory access
- timers
- semaphores
- traces
- partition supervision
- miscellaneous

4.5.1 Tasks

4.5.1.1 vsi_p_create () - Create Task

Function definition:

T_HANDLE vsi_p_create (T_HANDLE caller, SHORT (*pei_create)(T_PEI_INFO const ** info),
void (*TaskEntry)(USHORT, ULONG), T_HANDLE MemPoolHandle)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
SHORT(*pei_create)(T_PEI_INFO const **info)		address of pei_create() function	IN
void (*TaskEntry)(USHORT, ULONG)		address of task entry function	IN
T_HANDLE	MemPoolHandle	Identifier of memory pool to allocate task stack	IN

Return:

Type	Meaning	
T_HANDLE	VSI_ERROR	error
	handle	task handle

Options: none

Description:

The function vsi_p_create() is used to create a new task which pei_create() function is not entered in the task table in the configuration file xxxcomp.c.

The parameter *pei_create* is the address of the pei_create() function that exports all the parameters needed to create the task, refer to 4.4.1. The parameter *task_entry* is the entry point of the new task when it is scheduled the first time by the RTOS. If *task_entry* is set to NULL, then the common task entry function pf_TaskEntry of the frame is used.

The parameter *MemPoolHandle* is the handle of the memory pool from which the task stack will be allocated. For the creation of a new task it maybe needed to increase the size of the memory pool where the task stack is allocated from by setting INT_DATA_POOL_SIZE or EXT_DATA_POOL_SIZE in xxxconst.h to an appropriate value.

I executed successfully the function vsi_p_create() returns the handle of the new task.

The dynamically created task is not automatically started. This has to be done by the application by calling vsi_p_start().

4.5.1.2 vsi_p_delete () - Delete Task

Function definition:

SHORT vsi_p_delete (T_HANDLE Caller, T_HANDLE TaskHandle)

Parameters:

Type	Name	Meaning	
T_HANDLE	Caller	handle of calling entity	IN
T_HANDLE	TaskHandle	handle of task to be deleted	IN

Return:

Type	Meaning	
int	VSI_ERROR	error
	VSI_OK	success

Options: none

Description:

The function vsi_p_delete() is used to delete an existing task specified by the parameter *Taskhandle*.

Before the task is deleted the routing information for this task is cleared and its queue is deleted.

4.5.1.3 vsi_p_start () - Start Task

Function definition:

int vsi_p_start (T_HANDLE Caller, T_HANDLE TaskHandle)

Parameters:

Type	Name	Meaning	
T_HANDLE	Caller	handle of calling entity	IN
T_HANDLE	TaskHandle	handle of task to be started	IN

Return:

Type	Meaning	
int	VSI_ERROR	error
	VSI_OK	success

Options: none

Description:

The function vsi_p_start() is used to start the task specified by the parameter *Taskhandle* which has been dynamically created with vsi_p_create().

Tasks which pei_create() functions are entered in the component table in xxxcomp.c do not need to be started explicitly unless they are stopped with vsi_p_stop(), refer to 4.5.1.4.

4.5.1.4 vsi_p_stop () - Stop Task

Function definition:

int vsi_p_stop (T_HANDLE Caller, T_HANDLE TaskHandle)

Parameters:

Type	Name	Meaning	
T_HANDLE	Caller	handle of calling entity	IN
T_HANDLE	TaskHandle	handle of task to be stopped	IN

Return:

Type	Meaning	
int	VSI_ERROR	error
	VSI_OK	success

Options: none

Description:

The function vsi_p_stop() is used to stop the task specified by the parameter *Taskhandle*. A stopped task can be restarted by calling the function vsi_p_start().

4.5.1.5 vsi_p_name () - Get Task Name

Function definition:

int vsi_p_name (T_HANDLE Caller, T_HANDLE TaskHandle, char *Name)

Parameters:

Type	Name	Meaning	
T_HANDLE	Caller	handle of calling entity	IN
T_HANDLE	TaskHandle	task handle	IN
char *	Name	address of buffer to store the task name	OUT

Return:

Type	Meaning	
int	VSI_ERROR	error
	VSI_OK	success

Options: none

Description:

The function vsi_p_name() copies the name of the task specified by the parameter *TaskHandle* to the address passed in the parameter **Name*.

If the parameter *TaskHandle* is bigger than the maximum number of tasks (MAX_ENTITIES in xxxconst.h) or if no task with the passed *TaskHandle* is existing then VSI_ERROR is returned.

4.5.1.6 vsi_p_handle () - Get Task Handle

Function definition:

int vsi_p_handle (T_HANDLE Caller, char *Name)

Parameters:

Type	Name	Meaning	
T_HANDLE	Caller	handle of calling entity	IN
char *	Name	task name	IN

Return:

Type	Meaning	
int	VSI_ERROR	error
	handle	task handle

Options: none

Description:

The function vsi_p_handle() returns the handle of the task with the name specified by **Name*.

If no task with the Name specified by **Name* can by found then VSI_ERROR is returned.

If the parameter **Name* is set to NULL then the handle of the currently running task is returned.

4.5.1.7 vsi_p_exit () - Exit task

Function definition:

int vsi_p_exit (T_HANDLE Caller, T_HANDLE TaskHandle)

Parameters:

Type	Name	Meaning	
T_HANDLE	Caller	handle of calling entity	IN
T_HANDLE	TaskHandle	task handle	IN

Return:

Type	Meaning	
int	VSI_ERROR	error
	VSI_OK	success

Options: none

Description:

The function vsi_p_exit() is called to force the frame to call the pei_exit() function of the entity specified by the parameter *TaskHandle* to free all used resources. This is needed to be done before the corresponding task is deleted with vsi_p_delete(), refer to 4.5.1.2.

4.5.2 Entities

4.5.2.1 vsi_e_handle() – Get Entity Handle

Function definition:

T_HANDLE vsi_e_handle (T_HANDLE caller, char *name)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
char*	name	name of the requested entity	IN

Return:

Type	Meaning	
T_HANDLE	VSI_ERROR	error
	handle	entity handle

Options: none

Description:

The function vsi_e_handle () is used to request the handle of a previously created protocol stack entity.

When the parameter *name* is set to NULL, this function return the handle of the currently running entity.

4.5.2.2 vsi_e_name() - Get Entity Name

Function definition:

T_HANDLE vsi_e_name (T_HANDLE caller, T_HANDLE handle, char *name)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_HANDLE	handle	handle of requested	IN
char*	name	name of the requested entity	IN

Return:

Type	Meaning	
T_HANDLE	VSI_ERROR	error
	VSI_OK	success

Options: none

Description:

The function vsi_e_name () is used to request the name of a previously created protocol stack entity specified by it *handle.*

4.5.3 Communication

4.5.3.1 vsi_c_open () - Open Communication

Function definition:

T_HANDLE vsi_c_open (T_HANDLE caller, char *name)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
char*	name	name of the partner entity	IN

Return:

Type	Meaning	
T_HANDLE	VSI_ERROR	error
	handle	communication handle

Options: none

Description:

The function vsi_c_open () is used to open a previously created (by the frame) communication resource to the protocol stack entity with which the caller wishes to communicate. The function returns a handle that the caller may use for further access to the specified protocol stack entity.

4.5.3.2 vsi_c_close () - Close Communication

Function definition:

int vsi_c_close (T_HANDLE caller, T_HANDLE handle)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_HANDLE	name	handle of the partner entity	IN

Return:

Type	Meaning	
int	VSI_ERROR	error
	VSI_OK	success

Options: none

Description:

The function vsi_c_close () is used to close a previously opened communication resource.

4.5.3.3 vsi_c_clear () - Clear Communication Resource

Function definition:

int vsi_c_clear (T_HANDLE caller, T_HANDLE comhandle)

Parameters:

Type	Name	Meaning	Direction
T_HANDLE	caller	handle of calling entity	IN
T_HANDLE	comhandle	handle of the communication resource to be closed	IN

Return:

Type	Meaning	
int	VSI_ERROR	invalid communication handle
	VSI_OK	success

Options: none

Description:

The function vsi_c_clear () clears the communication resource specified by the handle i.e. queue entries are read and discarded until it is empty.

4.5.3.4 vsi_c_send () - Send Message

ATTENTION: THIS FUNCTION SHOULD NO LONGER BE USE BY VSI BASED SOFTWARE ENTITIES. PLEASE USE VSI_C_PSEND() INSTEAD

Function definition:

int vsi_c_send (T_HANDLE caller, T_HANDLE comhandle, T_QMSG * msg)

If partition pool monitoring activated:
int vsi_c_send (T_HANDLE caller, T_HANDLE comhandle, T_QMSG * msg, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_HANDLE	comhandle	destination queue handle	IN
T_QMSG *	msg	pointer to message to be sent	IN
const char *	file	file that called vsi_c_free() (PPM active)	IN
int	line	line where vsi_c_free() was called (PPM active)	IN

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid communication handle no communication resource available (non-task thread)

Options: MEMORY_SUPERVISION

Description:

The function vsi_c_send () writes the message buffer msg into the message queue a protocol stack entity specified by the communication handle. The message is a primitive, a signal or an timeout.

If the queue is full the calling task is suspended. If the request has been satisfied but the underlying OS-layer function had to wait for a free queue element an error message is traced by calling the function vsi_ttrace().

If the caller is a non-task (such as an ISR), the function returns immediately regardless if the request can be satisfied or not. In this case, VSI_ERROR is returned if no communication resource is available.

If the option MEMORY_SUPERVISION is set the function vsi_ppm_send() is called to supervise the state of the partition used to store the primitive.

Communication is carried out by copying the buffer content or by transmitting the buffer address (see 2.1.2).

4.5.3.5 vsi_c_psend () - Send Primitive

Function definition:

int vsi_c_psend (T_HANDLE comhandle, T_VOID_STRUCT *ptr, ULONG len)

If partition pool monitoring activated:
int vsi_c_psend (T_HANDLE comhandle, T_VOID_STRUCT *ptr, ULONG len, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	comhandle	destination queue handle	IN
T_VOID_STRUCT *	ptr	pointer to primitive to be sent	IN
ULONG *	len	length of primitive to be sent	IN
const char *	file	file that called vsi_c_psend() (PPM active)	IN
int	line	line where vsi_c_psend() was called (PPM active)	IN

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid communication handle
		no communication resource available (non-task thread)

Options: MEMORY_SUPERVISION

Description:

The function vsi_c_psend () calculates the handle of the calling entity and calls the function vsi_c_psend_caller() is called.

If you want to transmit a primitive use the macro PSEND instead of calling vsi_c_psend() directly (see 4.3.32).

4.5.3.6 vsi_c_psend_caller () - Send Primitive

Function definition:

int vsi_c_psend_caller (T_HANDLE caller, T_HANDLE comhandle, T_VOID_STRUCT *ptr, ULONG len)

If partition pool monitoring activated:
int vsi_c_psend_caller (T_HANDLE caller ,T_HANDLE comhandle, T_VOID_STRUCT *ptr, ULONG len, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	caller handle	IN
T_HANDLE	comhandle	destination queue handle	IN
T_VOID_STRUCT *	ptr	pointer to primitive to be sent	IN
ULONG *	len	length of primitive to be sent	IN
const char *	file	file that called vsi_c_psend() (PPM active)	IN
int	line	line where vsi_c_psend() was called (PPM active)	IN

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid communication handle
		no communication resource available (non-task thread)

Options: MEMORY_SUPERVISION

Description:

The function vsi_c_psend_caller () creates a message of the type T_QMSG and sets the element MsgType to MSG_PRIMITIVE, the element Prim to *ptr* and the element PrimLen to *len*. Then the function vsi_c_send() is called.

This function can be use to send from outside the GPF context and pretend to be a GPF entity, e.g. in callback functions.

If you want to transmit a primitive use the macro PSEND_CALLER instead of calling vsi_c_psend_caller() directly (see 4.3.33).

4.5.3.7 vsi_c_ssend () - Send Signal

Function definition:

int vsi_c_ssend (T_HANDLE comhandle, ULONG opc, T_VOID_STRUCT *ptr, ULONG len)

If partition pool monitoring activated:
int vsi_c_ssend (T_HANDLE comhandle, ULONG opc, T_VOID_STRUCT *ptr, ULONG len, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	comhandle	destination queue handle	IN
ULONG	opc	op-code of the signal to be sent	IN
T_VOID_STRUCT *	ptr	pointer to signal to be sent	IN
ULONG *	len	length of signal to be sent	IN
const char *	file	file that called vsi_c_psend() (PPM active)	IN
int	line	line where vsi_c_psend() was called (PPM active)	IN

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid communication handle
		no communication resource available (non-task thread)

Options: MEMORY_SUPERVISION

Description:

The function vsi_c_ssend () creates a message of the type T_QMSG and sets the element MsgType to MSG_SIGNAL, the element SigPtr to *ptr,* the element SigOPC to *opc* and the element SigLen to *len*. Then the function vsi_c_send() is called.

If you want to transmit a signal use the macro PSIGNAL instead of calling vsi_c_send() directly (see 4.3.36).

4.5.3.8 vsi_c_await() - Await primitive

Function definition:

int vsi_c_await (T_HANDLE caller, T_HANDLE comhandle, ULONG timeout , T_QMSG * msg,)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_HANDLE	comhandle	communication handle	IN
ULONG	timeout	time to wait for message in milliseconds	IN
T_QMSG *	msg	pointer to message to be received	IN

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid communication handle
	VSI_TIMEOUT	no message received

Options: only active Variant

Description:

The function vsi_c_await () is used to receive a primitive which has been sent by other components of the protocol stack.

The calling task is suspended until a message is received or the specified time expired. If the caller is a non-task, the function returns immediately regardless if the request can be satisfied or not. In this case, VSI_ERROR is returned if no message is received.

This function must not be called by the body in the passive variant of the protocol stack (see 2.1.1).

Communication is carried out by copying the buffer content or transmitting the buffer. The environment uses the parameter buf as target address for copying the incoming primitive or sets the parameter buf to the incoming buffer address of the primitive. The parameter type contains the message type, i.e. primitive, signal or timeout of the received message.

4.5.3.9 vsi_c_primitive () - Forward non GSM primitive

Function definition:

int vsi_c_primitive (T_HANDLE caller, void * prim)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
void *	prim	Primitive	IN

Return:

Type	Meaning	
int	VSI_OK	success

Options: none

Description:

The function vsi_c_primitive () is used to forward a configuration primitive to the environment.

If a protocol stack entity receives a primitive which is not a protocol related it is not processed, but rather forwarded to the frame for further evaluation.

4.5.3.10 vsi_c_new () - Allocate Partition Memory

ATTENTION: THIS FUNCTION SHOULD NO LONGER BE USE BY VSI BASED SOFTWARE ENTITIES. PLEASE USE VSI_C_PNEW() INSTEAD

Function definition:

T_VOID_STRUCT * vsi_c_new (T_HANDLE caller, ULONG size, ULONG opc)

If partition pool monitoring activated:

T_VOID_STRUCT * vsi_c_new (T_HANDLE caller, ULONG size, ULONG opc, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
ULONG	size	number of bytes needed	IN
ULONG	opc	primitive OPC	IN
const char *	file	file that called vsi_c_new()	IN
int	line	line where vsi_c_new() was called	IN

Return:

Type	Meaning	
void *	ptr	address of communication buffer
	NULL	no communication buffer available (non-task thread)

Options: None

Description:

The function vsi_c_new () is used to allocate a new partition for primitive communication from the partition memory pool. Therefor the function vsi_m_new () is called. If the number of requested bytes is smaller than the size of the primitive header then it is set to the size of the primitive header. The primitive header is initialized after successful allocation. The parameter *use_cnt* in the primitive header is set to one.

If no free partition is available at calling time the calling task is suspended. If the request has been satisfied but the underlying OS-layer function had to wait for a free partition a warning message is traced by calling the function vsi_ttrace().

If the caller is a non-task thread, and the request cannot be satisfied then an error message is traced and an RTOS/target specific error handling is performed.

This function is necessary if the communication is carried out by transmitting buffer addresses (see 2.1.2). It is not called if the communication is carried out by copying buffers.

4.5.3.11 vsi_c_pnew_generic () - Allocate Partition Memory

Function definition:

T_VOID_STRUCT * vsi_c_pnew_generic (T_HANDLE caller, ULONG size, ULONG opc, ULONG flags)

If partition pool monitoring activated:

T_VOID_STRUCT * vsi_c_pnew_generic (T_HANDLE caller, ULONG size, ULONG opc, ULONG flags, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
ULONG	size	number of bytes needed	IN
ULONG	opc	primitive OPC	IN
ULONG	flags	control pool selection and blocking behavior	IN
const char *	file	file that called vsi_c_pnew_generic()	IN
int	line	line where vsi_c_pnew_generic() was called	IN

Return:

Type	Meaning	
void *	ptr	address of communication buffer
	NULL	no communication buffer available (non-task thread)

Options: None

Description:

The function vsi_c_pnew_generic () is used to allocate a new partition for primitive communication from a partition memory pool defined by the parameter *flags*. The size parameter determines the number of bytes for user data, inside vsi_c_pnew_generic() the size of the primitive header is added. The primitive header is initialized after successful allocation. The parameter *use_cnt* in the primitive header is set to one.

If the bit VSI_MEM_NON_BLOCKING is set in the parameter *flags* a NULL pointer will be returned if no memory is available. If this is not set and no free partition is available at calling time the calling task is suspended. If the request has been satisfied but the underlying OS-layer function had to wait for a free partition a warning message is traced by calling the function vsi_ttrace().

If the caller is a non-task thread, and the request cannot be satisfied then an error message is traced and an RTOS/target specific fatal error handling is performed.

Please use the macro PALLOC_GENERIC instead of calling vsi_c_pnew_generic() directly (see 4.3.2).

4.5.3.12 vsi_c_pnew () - Allocate Partition Memory (blocking)

Function definition:

T_VOID_STRUCT * vsi_c_pnew (ULONG size, ULONG opc)

If partition pool monitoring activated:

T_VOID_STRUCT * vsi_c_pnew (ULONG size, ULONG opc, const char * file, int line)

Parameters:

Type	Name	Meaning	
ULONG	size	number of bytes needed	IN
ULONG	opc	primitive OPC	IN
const char *	file	file that called vsi_c_pnew()	IN
int	line	line where vsi_c_pnew() was called	IN

Return:

Type	Meaning	
void *	ptr	address of communication buffer
	NULL	no communication buffer available (non-task thread)

Options: None

Description:

The function vsi_c_pnew () is used to allocate a new partition for primitive communication from the protocol stack partition memory pool. The size parameter determines the number of bytes for user data, inside vsi_c_pnew() the size of the primitive header is added. The primitive header is initialized after successful allocation. The parameter *use_cnt* in the primitive header is set to one.

If no free partition is available at calling time the calling task is suspended. If the request has been satisfied but the underlying OS-layer function had to wait for a free partition a warning message is traced by calling the function vsi_ttrace().

If the caller is a non-task thread, and the request cannot be satisfied then an error message is traced and an RTOS/target specific fatal error handling is performed.

Please use the macro PALLOC instead of calling vsi_c_pnew() directly (see 4.3.1).

4.5.3.13 vsi_c_pnew_nb () - Allocate Partition Memory (non-blocking)

Function definition:

T_VOID_STRUCT * vsi_c_pnew_nb (ULONG size, ULONG opc)

If partition pool monitoring activated:

T_VOID_STRUCT * vsi_c_pnew_nb (ULONG size, ULONG opc, const char * file, int line)

Parameters:

Type	Name	Meaning	
ULONG	size	number of bytes needed	IN
ULONG	opc	primitive OPC	IN
const char *	file	file that called vsi_c_pnew_nb()	IN
int	line	line where vsi_c_pnew_nb() was called	IN

Return:

Type	Meaning	
void *	ptr	address of communication buffer
	NULL	no communication buffer available

Options: None

Description:

The function vsi_c_pnew_nb() has the same behavior as vsi_c_pnew() (4.5.3.12) except that it returns a NULL pointer in case the allocation request is not successful.

Please use the macro PALLOC_NB instead of calling vsi_c_pnew_nb() directly (see 4.3.1).

4.5.3.14 vsi_c_new_sdu () - Allocate Primitive containing an SDU

Function definition:

T_VOID_STRUCT * vsi_c_new_sdu (ULONG size, ULONG opc, USHORT sdu_len,
USHORT sdu_offset, USHORT encode_offset)

If partition pool monitoring activated:

T_VOID_STRUCT * vsi_c_new_sdu (ULONG Size, ULONG opc, USHORT sdu_len,
USHORT sdu_offset, USHORT encode_offset,
const char * file, int line)

Parameters:

Type	Name	Meaning	
ULONG	size	size in bytes of the primitive without SDU	IN
ULONG	opc	primitive opcode	IN
USHORT	sdu_len	length of the SDU in bits	IN
USHORT	sdu_offset	offset of T_sdu in primitive	IN
USHORT	encode_offset	coding offset in SDI in bits	IN
const char *	file	file that called vsi_c_pnew()	IN
int	line	line where vsi_c_pnew() was called	IN

Return:

Type	Meaning	
void *	ptr	address of communication buffer
	NULL	no communication buffer available (non-task thread)

Options: None

Description:

The function vsi_c_new_sdu () is serves as a wrapper for the function vsi_c_pnew() (4.5.3.10) to allocate memory partition from the protocol stack pool for a primitive containing an SDU. The amount of memory allocated is determined by the size of the primitive containing the SDU plus memory for the SDU specified by it length *sdu_len* and its offset *encode_offset*. Additionally the sdu pointer in the primitive header is initialized to point to the SDU in the primitive.

The returned pointer here points to the primitive data .

Use one of the macros PALLOC_MSG, PALLOC_SDU instead of calling vsi_c_new_sdu() directly (see 4.3.5).

4.5.3.15 vsi_c_new_sdu_generic () - Allocate Primitive containing an SDU

Function definition:

T_VOID_STRUCT * vsi_c_new_sdu (ULONG size, ULONG opc, USHORT sdu_len,
USHORT sdu_offset, USHORT encode_offset, ULONG flags)

If partition pool monitoring activated:

T_VOID_STRUCT * vsi_c_new_sdu (ULONG Size, ULONG opc, USHORT sdu_len,
USHORT sdu_offset, USHORT encode_offset, ULONG flags,
const char * file, int line)

Parameters:

Type	Name	Meaning	
ULONG	size	size in bytes of the primitive without SDU	IN
ULONG	opc	primitive opcode	IN
USHORT	sdu_len	length of the SDU in bits	IN
USHORT	sdu_offset	offset of T_sdu in primitive	IN
USHORT	encode_offset	coding offset in SDI in bits	IN
ULONG	flags	control the pool selection	IN
const char *	file	file that called vsi_c_pnew()	IN
int	line	line where vsi_c_pnew() was called	IN

Return:

Type	Meaning	
void *	ptr	address of communication buffer
	NULL	no communication buffer available (non-task thread)

Options: None

Description:

The function vsi_c_new_sdu_generic() has the same functionality as vsi_c_new_sdu() but the additional parameter *flags* allows the caller to select a partition pool to allocate from.

Use one of the macro FPALLOC_SDU instead of calling vsi_c_new_sdu_generic() directly (see 4.3.6).

4.5.3.16 vsi_c_ppass () - Pass Primitive Data to new Primitive

Function definition:

T_VOID_STRUCT * vsi_c_ppass (T_VOID_STRUCT *prim, ULONG new_opc)

If partition pool monitoring activated:

T_VOID_STRUCT * vsi_c_ppass (T_VOID_STRUCT *prim, ULONG new_opc,const char * file,int line)

Parameters:

Type	Name	Meaning	
T_VOID_STRUCT *	prim	pointer to data part of primitive	IN
ULONG	new_opc	new primitive opcode	IN
const char *	file	file that called vsi_c_pnew()	IN
int	line	line where vsi_c_pnew() was called	IN

Return:

Type	Meaning	
void *	ptr	pointer to data part of primitive

Options: MEMORY_SUPERVISION

Description:

The function vsi_c_ppass () enters the opcode specified by *new_opc* into the header of the primitive specified by *prim* and returns a pointer to the data part of this primitive.

In case the reference counter in the primitive header is bigger than one - this means that already a different user is attached to this primitive, e.g. the primitive is duplicated to PCO - a new memory partition is allocated, the original primitive content is copied to this memory, the *new_opc* is entered in the primitive header and a pointer to the data part of this new primitive is returned. This reallocation has to be done to avoid the modification of the header of a primitive that different entities are registered for.

It is recommended to use one of the macros PPASS or P_PASS (4.3.38) instead of calling vsi_c_ppass().

4.5.3.17 vsi_c_free () - Free Partition Memory

Function definition:

ATTENTION: THIS FUNCTION SHOULD NO LONGER BE USE BY VSI BASED SOFTWARE ENTITIES. PLEASE USE VSI_C_PFREE() INSTEAD

int vsi_c_free (T_HANDLE caller, T_VOID_STRUCT **addr)

If Partition Pool Monitor activated:
int vsi_c_free (T_HANDLE caller, T_VOID_STRUCT **addr, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_VOID_STRUCT **	addr	address of communication buffer	IN
const char *	file	file that called vsi_c_free()	IN
int	line	line where vsi_c_free() was called	IN

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid address of communication buffer

Options: MEMORY_SUPERVISION

Description:

The function vsi_c_free () releases a partition from the partition memory pool that was used for primitive communication. The pointer address is reset to the value NULL.

This function is used if the communication is carried out by transmitting buffer addresses (see 2.1.2) and is not used if the communication is carried out by copying buffers.

This function decrements the parameter *use_cnt* in the primitive header. If it is zero then the partition is freed by calling the function vsi_m_free().

4.5.3.18 vsi_c_pfree () - Free Primitive

Function definition:

int vsi_c_pfree (T_VOID_STRUCT **addr)

If Partition Pool Monitor activated:
int vsi_c_pfree (T_VOID_STRUCT **addr, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_VOID_STRUCT **	addr	address of communication buffer	IN
const char *	file	file that called vsi_c_pfree()	IN
int	line	line where vsi_c_pfree() was called	IN

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid address of communication buffer

Options: MEMORY_SUPERVISION

Description:

The function vsi_c_pfree () is serves as a wrapper for the function vsi_c_free(), see 4.5.3.14. It has been introduced to reduce the ROM size of the application. The functionality is the same as for vsi_c_free() but the parameter caller is omitted. The caller is set to zero and evaluated inside vsi_c_free() if really needed. The parameter **addr* here points to the primitive data. The address of the memory partition is also evaluated in vsi_c_pfree() and passed to vsi_c_free().

Use the macro PFREE instead of calling vsi_c_pfree() directly (see 4.3.15).

4.5.3.19 vsi_c_status () - Request Queue Status

Function definition:

int vsi_c_status (T_HANDLE q_handle, unsigned int *used, unsigned int *free);

Parameters:

Type	Name	Meaning	
T_HANDLE	q_handle	queue handle	IN
unsigned int *	used	number of messages in queue	OUT
unsigned int *	free	space in queue	OUT

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid q_handle

Options: none

Description:

The function vsi_c_status () return the number of message in a queue specified by its *q_handle* obtained by calling vsi_c_open().

4.5.3.20 vsi_c_pattach () - Attach to Primitive

Function definition:

int vsi_c_pattach (T_VOID_STRUCT *prim)

If partition memory supervision activated:
int vsi_c_pattach (T_VOID_STRUCT *prim, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_VOID_STRUCT *	prim	pointer to data part of the primitive to attach	IN

Return:

Type	Meaning	
int	VSI_OK	success

Options: MEMORY_SUPERVISION

Description:

The function vsi_c_pattach() allows the calling entity to attach to an already allocated primitive. This means the reference counter in the primitive header is incremented by one.

The attempt to attach to non-partition memory or to a partition that is not allocated results in a fatal error with the corresponding frame output via the test interface and a following reset of the phone.

It is recommended to call vsi_c_pattach via the macro PATTACH which will add the additional parameters if MEMORY_SUPERVISION is activated, refer to 4.3.14.

4.5.3.21 vsi_c_sync () - Synchronize with Protocol Stack

Function definition:

int vsi_c_sync (T_HANDLE caller, T_TIME timeout)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_TIME	timeout	timeout in ms for synchronization	IN

Return:

Type	Meaning	
int	VSI_OK	synchronization successful
	VSI_ERROR	no synchronization possible

Options: none

Description:

The function vsi_c_sync() is only available if the fame is compiled for the tool side.

The function vsi_c_sync() asks the TST entity in the tool environment to request the task states of the frame based tasks in the protocol stack. When all these tasks have completely started, vsi_c_sync() returns VSI_OK. As long not all tasks in the protocol stack have completely started, the synchronization attempts are performed until the time specified by the parameter *timeout* has occurred. In this case vsi_c_sync() returns with VSI_ERROR.

4.5.3.22 vsi_c_alloc_send () - Generic API Function to Send on Tool Side

Function definition:

int vsi_c_alloc_send (T_HANDLE com_handle, char* dst, char* src, void *prim, char *string)

Parameters:

Type	Name	Meaning	
T_HANDLE	com_handle	handle of destination entity	IN
char *	dst	name of destination entity in protocol stack	IN
char *	src	name of source entity on tool side	IN
void *	prim	pointer to allocated primitive to be sent	IN
char *	string	string to be sent to protocol stack	IN

Return:

Type	Meaning	
int	VSI_OK	data successfully sent
	VSI_ERROR	error

Options: none

Description:

The function vsi_c_alloc_send() is only available if the fame is compiled for the tool side.

The function vsi_c_alloc_send() is used to send any data from an application on the tool side to the protocol stack or any entity on the tool side specified by *com_handle*. If the destination entity is in the protocol stack, the *com_handle* has to be set to the test interface (TST) handle. In this case the parameter *dst* is mandatory.

The parameter *src* is only needed if the destination entity is interested in this.

If the parameter *prim* is different from NULL, it is assumed that the data to be sent is passed to vsi_c_alloc_send() in an already allocated primitive and *prim* points to the data part of this primitive.

If the parameter *string* is different from NULL, it is assumed that a string is passed to vsi_c_alloc_send(). In this case a memory partition is allocated and the *string* is copied into the data part of this primitive.

4.5.4 Memory

4.5.4.1 vsi_m_new () - Allocate Memory Partition

Function definition:

T_VOID_STRUCT * vsi_m_new (ULONG size, USHORT type)

If Partition Memory Monitor activated:
T_VOID_STRUCT * vsi_m_new (ULONG size, USHORT type, const char * file, int line)

Parameters:

Type	Name	Meaning	
ULONG	size	number of bytes needed	IN
USHORT	type	identifier for pool to allocate from	IN
const char *	file	file that called vsi_m_new()	IN
int	line	line where vsi_m_new() was called	IN

Return:

Type	Meaning	
T_VOID_STRUCT *	ptr	address of allocated buffer
	NULL	no buffer available (non-task thread)

Options: MEMORY_SUPERVISION

Description:

The function vsi_m_new () is used to allocate memory from a partition pool identified by the parameter *type*. If *type* is set to PRIM_POOL_PARTITION a partition is allocated from the primitive partition pool used for communication. If *type* is set to DMEM_POOL_PARTITION a partition is allocated from the non-communication partition pool. Allocation from application defined partition pool groups is also supported.

In addition to the partition pool group handle a set of flags is passed via the parameter *type.* If VSI_MEM_NONBLOCKING is set a NULL pointer will be returned if the addressed partition pool group is exhausted.

If VSI_MEM_NONBLOCKING is not set and no free partition is available at calling time the calling task is suspended. If the request has been satisfied but the underlying OS-layer function had to wait for a free partition a warning message is traced.

If the caller is a non-task thread, and the request cannot be satisfied then an error message is traced and an RTOS/target specific error handling is performed.

If the option MEMORY_SUPERVISION is set the function vsi_ppm_new() is called to supervise the state of the partition.

4.5.4.2 vsi_m_cnew () - Allocate Memory Partition

Function definition:

T_VOID_STRUCT * vsi_m_cnew (ULONG size, USHORT type)

If Partition Memory Monitor activated:
T_VOID_STRUCT * vsi_m_cnew (ULONG size, USHORT type, const char * file, int line)

Parameters:

Type	Name	Meaning	
ULONG	size	number of bytes needed	IN
USHORT	type	identifier for pool to allocate from	IN
const char *	file	file that called vsi_m_cnew()	IN
int	line	line where vsi_m_cnew() was called	IN

Return:

Type	Meaning	
T_VOID_STRUCT *	ptr	address of allocated buffer
	NULL	no buffer available (non-task thread)

Options: MEMORY_SUPERVISION

Description:

The function vsi_m_cnew () is used to allocate memory from a partition pool identified by the parameter *type*. If *type* is set to PRIM_POOL_PARTITION a partition is allocated from the primitive partition pool used for communication. If *type* is set to DMEM_POOL_PARTITION a partition is allocated from the non-communication partition pool. Allocation from application defined partition pool groups is also supported.

In addition to the partition pool group handle a set of flags is passed via the parameter *type.* If VSI_MEM_NONBLOCKING is set a NULL pointer will be returned if the addressed partition pool group is exhausted.

If VSI_MEM_NONBLOCKING is not set and no free partition is available at calling time the calling task is suspended. If the request has been satisfied but the underlying OS-layer function had to wait for a free partition a warning message is traced.

If the caller is a non-task thread, and the request cannot be satisfied then an error message is traced and an RTOS/target specific error handling is performed.

If the option MEMORY_SUPERVISION is set the function vsi_ppm_new() is called to supervise the state of the partition.

Use the macro MALLOC instead of calling vsi_m_cnew() directly (see 4.3).

4.5.4.3 vsi_m_realloc () - Realloc Memory Partition

Function definition:

T_VOID_STRUCT * vsi_m_realloc (T_VOID_STRUCT* ptr0, ULONG size, USHORT type)

If Partition Memory Monitor activated:
T_VOID_STRUCT * vsi_m_realloc (T_VOID_STRUCT* ptr0 ,ULONG size, USHORT type, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_VOID_STRUCT *	ptr0	pointer to allocated memory	IN
ULONG	size	number of bytes needed	IN
USHORT	type	identifier for pool to allocate from	IN
const char *	file	file that called vsi_m_realloc()	IN
int	line	line where vsi_m_realloc() was called	IN

Return:

Type	Meaning	
T_VOID_STRUCT *	ptr	address of reallocated buffer
	NULL	no buffer available

Options: MEMORY_SUPERVISION

Description:

The function vsi_m_realloc () is used to reallocate memory from a partition pool identified by the parameter *type*.

If the allocated memory block specified by *ptr0* is sufficiently large to store *size* bytes, *ptr0* is returned. If the allocated block specified by *ptr0* is too small, a memory block big enough to store *size* bytes is allocated, the contents of *ptr0* is copied to the newly allocated block, the block specified by *ptr0* is deallocated and the address of the allocated memory block is returned. Memory is allocated by calling the function vsi_m_cnew() (see 4.5.4.2). If *ptr0* is NULL a memory block of *size* bytes is allocated. If *size* is equal to zero and *ptr0* is not NULL, the memory block specified by *ptr0* is freed.

If *type* is set to PRIM_POOL_PARTITION a partition is allocated from the primitive partition pool used for communication. If *type* is set to DMEM_POOL_PARTITION a partition is allocated from the non-communication partition pool. Allocation from application defined partition pool groups is also supported.

In addition to the partition pool group handle a set of flags is passed via the parameter *type.* If VSI_MEM_NONBLOCKING is set a NULL pointer will be returned if the addressed partition pool group is exhausted.

If VSI_MEM_NONBLOCKING is not set and no free partition is available at calling time the calling task is suspended. If the request has been satisfied but the underlying OS-layer function had to wait for a free partition a warning message is traced.

If the caller is a non-task thread, and the request cannot be satisfied then an error message is traced and an RTOS/target specific error handling is performed.

If the option MEMORY_SUPERVISION is set the function vsi_ppm_new() is called to supervise the state of the partition.

Use the macro M(D)REALLOC(_NB) or M(D)_REALLOC(_NB) instead of calling vsi_m_realloc() directly (see 4.3).

4.5.4.4 vsi_m_free () - Free Memory

Function definition:

int vsi_m_free (T_VOID_STRUCT**addr)

If partition memory supervision activated:
int vsi_m_free (T_VOID_STRUCT**addr, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_VOID_STRUCT **	addr	address of buffer to be released	IN
const char *	file	file that called vsi_m_free()	IN
int	line	line where vsi_m_free() was called	IN

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid buffer address

Options: MEMORY_SUPERVISION, OPTIMIZE_POOL

Description:

The function vsi_m_free() deallocates a memory partition. The pointer address is reset to the value NULL.

If the memory to be freed is already freed or if the passed pointer does not point to a memory partition a warning message is traced if the this information is returned from the OS adaptation layer.

If the option MEMORY_SUPERVISION is set the function vsi_ppm_free() is called to supervise the state of the partition used to store the primitive.

Use the macro MFREE instead of calling vsi_m_free() directly (see 4.3).

4.5.4.5 vsi_m_status () - Get Memory Status

Function definition:

int vsi_m_status (T_HANDLE caller, ULONG size, USHORT type, USHORT *available , USHORT *allocated)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
ULONG	size	requested size	IN
USHORT	type	identifier for partition pool	IN
USHORT *	available	number of free partitions	OUT
USHORT *	allocated	number of allocated partitions	OUT

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid partition size requested

Options: ---

Description:

The function vsi_m_status() returns the number of available and allocated partitions in a pool specified by the parameters *type* and *size* .

4.5.4.6 vsi_m_attach () - Attach to Memory

Function definition:

int vsi_m_attach (T_VOID_STRUCT *mem)

If partition memory supervision activated:
int vsi_m_attach (T_VOID_STRUCT *mem, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_VOID_STRUCT *	mem	pointer to memory to attach	IN

Return:

Type	Meaning	
int	VSI_OK	success

Options: MEMORY_SUPERVISION

Description:

The function vsi_m_attach() allows the calling entity to attach to an already allocated memory partition. This means the reference counter in the partition header is incremented by one.

The attempt to attach to non-partition memory or to a partition that is not allocated results in a fatal error with the corresponding frame output via the test interface and a following reset of the phone.

It is recommended to call vsi_m_attach via the macro MATTACH which will add the additional parameters if MEMORY_SUPERVISION is activated, refer to 4.3.28.

4.5.4.7 vsi_drpo_new () - Allocate Root of Dynamic Primitive

Function definition:

T_VOID_STRUCT * vsi_drpo_new (ULONG size, ULONG opc, ULONG guess)

If Partition Memory Monitor activated:
T_VOID_STRUCT * vsi_drpo_new (ULONG size, ULONG opc, ULONG guess, const char * file, int line)

Parameters:

Type	Name	Meaning	
ULONG	size	number of bytes needed	IN
ULONG	opc	op-code of the primitive	IN
ULONG	guess	additionally required space in partition	IN
const char *	file	file that called vsi_drpo_new()	IN
int	line	line where vsi_drpo_new() was called	IN

Return:

Type	Meaning	
T_VOID_STRUCT *	ptr	address of allocated buffer

Options: MEMORY_SUPERVISION

Description:

The function vsi_drpo_new () is used to allocate the root of a dynamic sized primitive, refer to [C_8434.100]. The parameter *size* defines the number of bytes to by stored at the current allocation. The parameter *guess* can be used to reserve memory for future allocations in the same partition. If guess is set to a value bigger than zero then a partition to store *size* plus *guess* bytes is allocated. If the caller has no idea of the size of subsequent allocations then *guess* can be set to zero. In this case three times the size of the parameter *size* is allocated.

The function vsi_drpo_new() returns a pointer to the user data of the primitive. This pointer has to be passed to vsi_dp_new() for the allocation of additional memory in an existing dynamic primitive.

If no free partition is available at calling time the calling task is suspended. If the request has been satisfied but the underlying OS-layer function had to wait for a free partition a warning message is traced.

If the caller is a non-task thread, and the request cannot be satisfied then an error message is traced and an RTOS/target specific error handling is performed.

Use the macro DRPO_ALLOC instead of calling vsi_drpo_new() directly (see 4.3.8).

4.5.4.8 vsi_drp_new () - Allocate Root of Dynamic Memory

Function definition:

T_VOID_STRUCT * vsi_drp_new (ULONG size, ULONG guess)

If Partition Memory Monitor activated:
T_VOID_STRUCT * vsi_drp_new (ULONG size, ULONG guess, const char * file, int line)

Parameters:

Type	Name	Meaning	
ULONG	size	number of bytes needed	IN
ULONG	guess	additionally required space in partition	IN
const char *	file	file that called vsi_drp_new()	IN
int	line	line where vsi_drp_new() was called	IN

Return:

Type	Meaning	
T_VOID_STRUCT *	ptr	address of allocated buffer

Options: MEMORY_SUPERVISION

Description:

The function vsi_drp_new () is used to allocate the root of a dynamic sized memory, refer to [C_8434.100]. The parameter *size* defines the number of bytes to by stored at the current allocation. The parameter *guess* can be used to reserve memory for future allocations in the same partition. If guess is set to a value bigger than zero then a partition to store *size* plus *guess* bytes is allocated. If the caller has no idea of the size of subsequent allocations then *guess* can be set to zero. In this case three times the size of the parameter *size* is allocated.

The dynamic sized memory allocated with vsi_drp_new() must not be confused with the dynamic sized primitive allocated with vsi_drpo_new(). Dynamic sized primitives contain an operation code (opc) and therefor can be sent to different entities, the dynamic sized memory allocated with vsi_drp_new() cannot be sent through the stack as a primitive.

The function vsi_drp_new() returns a pointer to the user data of the primitive. This pointer has to be passed to vsi_dp_new() for the allocation of additional memory in an existing dynamic primitive.

If no free partition is available at calling time the calling task is suspended. If the request has been satisfied but the underlying OS-layer function had to wait for a free partition a warning message is traced.

If the caller is a non-task thread, and the request cannot be satisfied then an error message is traced and an RTOS/target specific error handling is performed.

Use the macro DRP_ALLOC instead of calling vsi_drp_new() directly (see 4.3.9).

4.5.4.9 vsi_drp_bind () - Bind child root-pointer to parent root-pointer

Function definition:

int vsi_drp_bind (T_VOID_STRUCT *child, T_VOID_STRUCT *parent)

If Partition Memory Monitor activated:
int vsi_drp_binb (T_VOID_STRUCT *child, T_VOID_STRUCT *parent, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_VOID_STRUCT *	child	pointer to child root pointer	IN
T_VOID_STRUCT *	parent	pointer to parent root pointer	IN
const char *	file	file that called vsi_drp_new()	IN
int	line	line where vsi_drp_new() was called	IN

Return:

Type	Meaning	
int	status	VSI_OK or VSI_ERROR

Options: MEMORY_SUPERVISION

Description:

The function vsi_drp_bind () is used to bind the root of a dynamic sized memory (refer to [C_8434.100]) to another root.

The parent has to allocated with DRPO_ALLOC or DRP_ALLOC. The child has to be allocated with DRPO_ALLOC, DRP_ALLOC or one of the PALLOC-like macros. vsi_drp_bind () adds child to the internal drp_bound_list of parent and (recursively) increases the child use_cnt. When FREE is called for parent FREE is also called for all bound childs.

If the binding succeeded VSI_OK is returned.

If no more entries are available (see MAX_DRP_BOUND) in the drp_bound_list or the drp_bound_list could not be allocated VSI_ERROR is returned.

Use the macro DRP_BIND instead of calling vsi_drp_bind() directly (see 4.3.10).

4.5.4.10 vsi_dp_new () - Allocate Additional Dynamic Memory

Function definition:

T_VOID_STRUCT * vsi_dp_new (ULONG size, T_VOID_STRUCT *ptr, ULONG guess)

If Partition Memory Monitor activated:
T_VOID_STRUCT * vsi_dp_new (ULONG size, T_VOID_STRUCT *ptr, ULONG guess, const char * file, int line)

Parameters:

Type	Name	Meaning	
ULONG	size	number of bytes needed	IN
T_VOID_STRUCT *	ptr	pointer to root of dynamic primitive/memory	IN
ULONG	guess	additionally required space	IN
const char *	file	file that called vsi_dp_new()	IN
int	line	line where vsi_dp_new() was called	IN

Return:

Type	Meaning	
T_VOID_STRUCT *	ptr	address of allocated buffer

Options: MEMORY_SUPERVISION

Description:

The function vsi_dp_new () is used to allocate additional memory in an already allocated partition respectively chain of memory/dynamic primitive, refer to [C_8434.100]. The parameter *size* defines the number of bytes to by stored at the current allocation. The parameter *guess* can be used to reserve memory for future allocations. If guess is set to a value bigger than zero then a partition to store *size* plus *guess* bytes is allocated. If the caller has no idea of the size of subsequent allocations then *guess* can be set to zero. In this case three times the size of the parameter *size* is allocated.

The parameter *ptr* that has to be passed to the vsi_dp_new() is the pointer returned by vsi_drpo_new(9 or vsi_drp_new(). The function vsi_dp_new() returns a pointer to the user data of the allocated memory.

If no free partition is available at calling time the calling task is suspended. If the request has been satisfied but the underlying OS-layer function had to wait for a free partition a warning message is traced.

If the caller is a non-task thread, and the request cannot be satisfied then an error message is traced and an RTOS/target specific error handling is performed.

Use the macro DP_ALLOC instead of calling vsi_dp_new() directly (see 4.3.11).

4.5.4.11 vsi_dp_sum () - Get Number of Stored Bytes in Dynamic Memory

Function definition:

ULONG vsi_dp_sum (T_HANDLE caller, T_VOID_STRUCT **addr)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_VOID_STRUCT *	addr	address of communication buffer	IN
ULONG *	size	number of stored bytes in chain	OUT

Return:

Type	Meaning	
ULONG	VSI_OK	success
	VSI_ERROR	invalid parameter **addr*

Options: none

Description:

The function vsi_dp_sum() returns the number of stored bytes in a chain of allocated memory/dynamic primitive.

4.5.4.12 vsi_free () - Free Dynamic Sized Memory

Function definition:

int vsi_free (T_VOID_STRUCT **addr)

If Partition Pool Monitor activated:
int vsi_free (T_VOID_STRUCT **addr, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_VOID_STRUCT **	addr	address of communication buffer	IN
const char *	file	file that called vsi_free()	IN
int	line	line where vsi_free() was called	IN

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid address of communication buffer

Options: MEMORY_SUPERVISION

Description:

The function vsi_free() releases a memory partition respectively a chain of allocated memory/dynamic primitive. The pointer address is reset to the value NULL. It deallocates partitions allocated with vsi_c_new(), vsi_drpo_new() and vsi_drp_new(). To deallocate a chain of dynamic memory the parameter *addr* has to be the root of the chain.

This function decrements the parameter *use_cnt* in the primitive header. If it is zero then the partition is freed by calling the function vsi_m_free().

Use the macro FREE instead of calling vsi_free() directly (see 4.3.29).

4.5.5 Virtual Memory Pools

4.5.5.1 vsi_vmp_create () - Create Virtual Memory Pool

Function definition:

int vsi_vmp_create (T_HANDLE * vmp_handle, T_VSI_VMP_CONFIG_PARAM * vmp_config
T_VSI_VMP_NOTIFY_PARAM * vmp_notify, U8 * name)

Parameters:

Type	Name	Meaning	
T_HANDLE *	vmp_handle	handle of virtual memory pool	OUT
T_VSI_VMP_CONFIG_PARAM *	vmp_config	configuration parameter	IN
T_VSI_VMP_NOTIFY_PARAM *	vmp_notify	notification parameter	IN
U8 *	name	name for the virtual memory pool	IN

Return:

Type	Meaning
int	VSI_OK VSI_INVALID_PARAM VSI_NO_MEMORY

Options: none

Description:

The function vsi_vmp_create() is used to create a virtual memory pool with the properties specified in **vmp_config*, see 4.1.9). Such a pool can be used by any GSP entity in the system to which a certain amount of memory (which must not be exceeded) is assigned. For flow control usage the entity has to set the appropriate values in **vmp_config*. For notification about flow control changes the entity has to supply an entity handle and/or a callback function.

The *size* element in the attributes determines the number of bytes that a client of the virtual memory pool can allocate in sum.

If the VSI_FAST_MEMORY is set in the flags of **vp_config* the virtual memory pool will be created in the fast memory region, e.g. internal RAM (if available).

The element *flags* in *T_VSI_VMP_CONFIG_PARAM* also controls the blocking behavior at allocation time.

The attempt to create a virtual memory pool bigger than the available physical memory in the specified memory region will result in returning VSI_NO_MEMORY.

The attempt to create an additional virtual memory pool that will result in all virtual memory pool sizes together exceed the size of the specified memory region will result in returning VSI_NO_MEMORY.

In case of an error the parameter **vmp_handle* will be set to zero.

4.5.5.2 vsi_vmp_delete () - Delete Virtual Memory Pool

Function definition:

int vsi_vmp_delete (T _HANDLE vmp_handle)

Parameters:

Type	Name	Meaning	
T_HANDLE	vmp_handle	handle of virtual memory pool	IN

Return:

Type	Meaning
int	VSI_OK VSI_INVALID_PARAM VSI_VMP_NOT_CLEAN

Options: none

Description:

The function vsi_vmp_delete() is used to delete a virtual memory pool specified by the handle *vmp_handle.*

The warning code VSI_VMP_NOT_CLEAN is returned if the virtual memory pool has not been cleaned up before deletion, i.e. there still are allocations in the memory pool.

ATTENTION: Task that are suspended while waiting for memory in the specified pool will be resumed and the memory allocation API function will return a NULL pointer (Nucleus RTOS). This behaviour may be different for other operating systems.

ATTENTION: Entities waiting flow a change of the flow control status to off (memory available) will not be informed about the pool deletion.

The error code VSI_INVALID_HANDLE is returned if no virtual memory pool with the specified handle exists in the GSP context.

4.5.5.3 vsi_vmp_modify () - Modify Virtual Memory Pool

Function definition:

int vsi_vmp_modify (T_HANDLE vmp_handle, T_VSI_VMP_CONFIG_PARAM * vmp_config
T_VSI_VMP_NOTIFY_PARAM * vmp_notify)

Parameters:

Type	Name	Meaning	
T_HANDLE	vmp_handle	handle of virtual memory pool	OUT
T_VSI_VMP_CONFIG_PARAM *	vmp_config	configuration parameter	IN
T_VSI_VMP_NOTIFY_PARAM *	vmp_notify	notification parameter	IN

Return:

Type	Meaning
int	VSI_OK VSI_INVALID_PARAM

Options: none

Description:

The function vsi_vmp_modify() is used to change parameters of a virtual memory pool specified by the handle *vmp_handle.*

The behaviour is like the vsi_vmp_create () function, see 4.5.5.1), except of . *vmp_handle* is an input parameter and no virtual memory pool is created.

The error code VSI_INVALID_HANDLE is returned if no virtual memory pool with the specified handle exists in the GSP context.

4.5.5.4 vsi_vmp_get_status () - Get Virtual Memory Pool Status

Function definition:

int vsi_vmp_get_status (T_HANDLE vmp_handle, unsigned int *free, unsigned int *alloc,
int *flow_ctrl_state)

Parameters:

Type	Name	Meaning	
T_HANDLE	vmp_handle	handle of virtual memory pool	IN
unsigned int *	free	amount of available bytes	OUT
unsigned int *	alloc	amount of allocated bytes	OUT
int *	flow_ctrl_state	flow control state	OUT

Return:

Type	Meaning
int	VSI_OK VSI_INVALID_PARAM

Options: none

Description:

The function vsi_vmp_get_status() returns the status of the virtual memory pool specified by *vmp_handle*. The amount of available bytes is returned in **free*, and the amount of allocated bytes in **alloc*. Flow control state on is indicated by **flow_ctrl_state* set to VSI_VMP_FLOW_CTRL_ON, flow control off by VSI_VMP_FLOW_CTRL_OFF.

The error code VSI_INVALID_PARAM is returned if no virtual memory pool with the specified handle exists in the GSP context.

4.5.5.5 vsi_vmp_malloc() - Allocate Memory Block from Virtual Memory Pool

Function definition:

T_VOID_STRUCT* vsi_vmp_malloc(T_HANDLE vmp_handle, T_VSI_SIZE size)

If memory supervision is enabled:
T_VOID_STRUCT* vsi_vmp_malloc (T_HANDLE vp_handle, T_VSI_SIZE size,
const char * file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	vmp_handle	handle of virtual pool	IN
T_VSI_SIZE	size	number of bytes to allocate	IN

Return:

Type	Meaning	
T_VOID_STRUCT	!= NULL	success
	NULL	allocation failed

Options: memory supervision

Description:

The function vsi_vmp_malloc() is called allocate a block of *size* bytes from the virtual memory pool specified by *vmp_handle.* The allocation will be non-blocking, i.e. a NULL pointer will be returned if no memory is available.

A change of the flow control state to “on” will be signalled to the caller if the number of allocated bytes in the virtual pool exceeds the *on_level* passed to vsi_vmp_create() or vsi_vmp_modify().

An invalid *vmp_handle* will result in a NULL pointer returned by vsi_vmp_malloc().

ATTENTION: In the first version of the virtual memory pool handling only the non-blocking allocation service provided by the existing macro/function call MALLOC_NB()/vsi_m_cnew() will be offered. Allocation function for primitives (PALLOC) dynamic primitive (DRPO_ALLOC) may follow on request.

4.5.5.6 vsi_vmp_mfree () - Return Memory Block

Function definition:

int vsi_vmp_mfree (T_VOID_STRUCT * mem)

If memory supervision is enabled:
int vsi_virtual_pool_mfree (T_VOID_STRUCT * mem, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_VOID_STRUCT *	mem	memory block to deallocate	IN

Return:

Type	Meaning
int	VSI_OK VSI_ERROR

Options: memory supervision

Description:

The function vsi_vmp_free() is called to deallocate a block of memory previously allocated from a virtual memory pool.

A change of the flow control state to “off” will be signalled to the caller if the number of allocated bytes in the virtual pool falls below the *off_level* passed to vsi_vmp_create() or vsi_vmp_modify().

4.5.5.7 vsi_vmp_notify_split () - Notify about split Memory Block

Function definition:

int vsi_vmp_notify_split (T_VOID_STRUCT * mem)

If memory supervision is enabled:
int vsi_virtual_pool_mfree (T_VOID_STRUCT * mem, const char * file, int line)

Parameters:

Type	Name	Meaning	
T_VOID_STRUCT *	mem	memory block which is split	IN

Return:

Type	Meaning
int	VSI_OK VSI_ERROR

Options: memory supervision

Description:

The function vsi_vmp_notify_split() is called to notify the virtual pool manager about a split of a block of memory previously allocated from a virtual memory pool.

4.5.6 Timer

4.5.6.1 vsi_t_start () - Start Timer

Function definition:

int vsi_t_start (T_HANDLE caller, USHORT index, T_TIME value)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
USHORT	index	index of timer	IN
T_TIME	value	time in ms	IN

Return:

Type	Meaning	
int	VSI_ERROR	invalid timer index
	VSI_OK	success

Options: OPTION_TIMER_CONFIG

Description:

The function vsi_t_start () starts an application timer with the specified time in milliseconds.

The index is stored when the timer is started and forwarded to the protocol stack entity by the function pei_timeout() when the timer has expired. The timer is started only once. The maximum timer value to be passed to vsi_t_start() is 4 294 967 295 ms.

The index of the timer is the one defined in the corresponding protocol stack entity and is in the range from zero to the number of timers minus one for the corresponding entity defined in pei_create(). If a timer index out of this range is passed to the function, then a error message is traced and the active task is suspended forever.

A timeout is forwarded to the corresponding entity through its message queue with the same priority like primitives. This prevents an ongoing processing for a primitive from being interrupted by a timeout.

Timers can be dynamically configured with configuration primitives received via the test interface, refer to 4.5.6.5.

4.5.6.2 vsi_t_pstart () - Start Timer with Periodic Reload

Function definition:

int vsi_t_pstart (T_HANDLE caller, USHORT index, T_TIME value1, T_TIME value2)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
USHORT	index	index of timer	IN
T_TIME	value1	initial time in ms	IN
T_TIME	value2	reschedule time in ms	IN

Return:

Type	Meaning	
int	VSI_ERROR	invalid timer index
	VSI_OK	success

Options: OPTION_TIMER_CONFIG

Description:

The function vsi_t_pstart () starts an application timer with the specified initial time in milliseconds. After expiration, the timer is started periodically with the rescheduling time until it is stopped.

The index is stored when the timer is started and forwarded to the protocol stack entity by the function pei_timeout() when a the timer has expired. The maximum timer values to be passed to vsi_t_pstart() are 4 294 967 295 ms.

The index of the timer is the one defined in the corresponding protocol stack entity and is in the range from zero to the number of timers minus one for the corresponding entity defined in pei_create(). If a timer index out of this range is passed to the function, then a error message is traced and the active task is suspended forever.

A timeout is forwarded to the corresponding entity through its message queue with the same priority like primitives. This prevents an ongoing processing for a primitive from being interrupted by a timeout.

Timers can be dynamically configured with configuration primitives received via the test interface, refer to 4.5.6.5.

4.5.6.3 vsi_t_stop () - Stop Timer

Function definition:

int vsi_t_stop (T_HANDLE caller, USHORT index)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
USHORT	index	index of timer	IN

Return:

Type	Meaning	
int	VSI_ERROR	invalid timer index
	VSI_OK	success

Description:

The function vsi_t_stop () causes the timer specified by the calling entity and the index to stop even if the time has not expired.

4.5.6.4 vsi_t_status () - Query Timer Status

Function definition:

int vsi_t_status (T_HANDLE caller, USHORT index, T_TIME * tvalue)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
USHORT	index	index of timer	IN
T_TIME *	tvalue	remaining time until timer expires in ms	OUT

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid timer handle

Options: none

Description:

The function vsi_t_status () stores the remaining time of the timer specified by the calling entity and the index in the tvalue.

4.5.6.5 vsi_t_config () - Configure Timer

Function definition:

int vsi_t_config (T_HANDLE caller, USHORT index, USHORT mode, T_TIME tvalue)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
USHORT	index	index of timer	IN
USHORT	mode	mode of timer	IN
T_TIME	tvalue	expiration time in ms	IN

Return:

Type	Meaning	
int	VSI_OK	success
	VSI_ERROR	invalid timer handle

Options: OPTION_TIMER_CONFIG

Description:

The function vsi_t_config() enables dynamic configuration to manipulate timer values (set to new value, suppress timer, slow down or speed up timer).

This function stores the dynamic configuration data for the specified timer in a configuration table. When the timer is started, the parameters of this table are used. The different modes are defined as follows:

TIMER_SET: Start with tvalue.
TIMER_RESET: Start with original timer value.
TIMER_SPEED_UP: Start with original time divided by tvalue.
TIMER_SLOW_DOWN:Start with original time multiplied by tvalue.

4.5.6.6 vsi_t_time () - Query System Clock

Function definition:

int vsi_t_time (T_HANDLE caller, T_TIME * tvalue)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_TIME *	tvalue	time in ms	OUT

Return:

Type	Meaning	
int	VSI_OK	success

Options: none

Description:

The function vsi_t_time () stores the time in milliseconds since system start in tvalue.

4.5.6.7 vsi_t_sleep () - Suspend Thread

Function definition:

int vsi_t_sleep (T_HANDLE caller, T_TIME tvalue)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_TIME	tvalue	time in ms	IN

Return:

Type	Meaning	
int	VSI_OK	success

Options: none

Description:

The function vsi_t_sleep () suspends the calling task for the time specified by tvalue.

4.5.7 Semaphores

4.5.7.1 vsi_s_open () - Open Semaphore

Function definition:

T_HANDLE vsi_s_open (T_HANDLE caller, char * name, USHORT count)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
char *	name	name of semaphore	IN
USHORT	count	initial count of the semaphore (e.g. 1 for a binary sem.)	IN

Return:

Type	Meaning	
T_HANDLE	VSI_ERROR	error
	handle	handle of opened semaphore

Options: none

Description:

The function vsi_s_open() opens a (counting) semaphore specified by its name. If the semaphore does not exist, it will be created with the initial count given. If the semaphore already exists the parameter count will be ignored.

4.5.7.2 vsi_s_close () - Close Semaphore

Function definition:

int vsi_s_close (T_HANDLE caller, T_HANDLE handle)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_HANDLE	handle	handle of semaphore	IN

Return:

Type	Meaning	
int	VSI_ERROR	invalid handle of semaphore
	VSI_OK	success

Options: none

Description:

The function vsi_s_close() closes a semaphore.

4.5.7.3 vsi_s_get () - Get Semaphore

Function definition:

int vsi_s_get (T_HANDLE caller, T_HANDLE handle)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_HANDLE	handle	handle of semaphore	IN

Return:

Type	Meaning	
int	VSI_ERROR	invalid handle of semaphore
	VSI_OK	success

Options: none

Description:

The function vsi_s_get() obtains the specified semaphore, i.e. the counter is decremented, if it is greater than zero.

If the counter is equal to zero, than the calling task is suspended until the counter is incremented by another task (vsi_s_release()).

If the caller is a non-task thread the function returns immediately regardless if the request can be satisfied or not. In this case, VSI_ERROR is returned if the counter was already zero.

4.5.7.4 vsi_s_get_timeout () - Get Semaphore with Timeout

Function definition:

int vsi_s_get_timeout (T_HANDLE caller, T_HANDLE handle, T_TIME timeout)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_HANDLE	handle	handle of semaphore	IN
T_TIME	timeout	timeout in ms	IN

Return:

Type	Meaning	
int	VSI_ERROR	invalid handle of semaphore
	VSI_OK	success

Options: none

Description:

The function vsi_s_get_timeout() obtains the specified semaphore, i.e. the counter is decremented, if it is greater than zero.

If the counter is equal to zero, than the calling task is suspended until the counter is incremented by another task (vsi_s_release()) or the time in ms specified by the parameter *timeout* has passed. In case the specified time has passed without the semaphore becoming available the function returns VSI_TIMEOUT.

If the caller is a non-task thread the function returns immediately regardless if the request can be satisfied or not. In this case, VSI_ERROR is returned if the counter was already zero.

4.5.7.5 vsi_s_release () - Release Semaphore

Function definition:

int vsi_s_release (T_HANDLE caller, T_HANDLE handle)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_HANDLE	handle	handle of semaphore	IN

Return:

Type	Meaning	
int	VSI_ERROR	invalid handle of semaphore
	VSI_OK	success

Options: none

Description:

The function vsi_s_release () releases the specified semaphore, i.e. the counter is incremented.

4.5.7.6 vsi_s_status () - Query Semaphore Counter Value

Function definition:

int vsi_s_status (T_HANDLE caller, T_HANDLE handle, USHORT * count)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_HANDLE	handle	handle of semaphore	IN
USHORT *	count	current value of semaphore counter	OUT

Return:

Type	Meaning	
int	VSI_ERROR	invalid handle of semaphore
	VSI_OK	success

Options: none

Description:

The function vsi_s_status () can be used to obtain the counter of a semaphore.

4.5.8 Traces

The tracing functionality includes some filtering of messages to be traced. Due to this feature, it is possible to select a certain number of all implemented traces to be disabled. To support this, different trace classes e.g. TC_PRIM, TC_STATE are installed, refer to 4.2.3. A trace class is defined by one bit in a mask. This trace class is passed to the specific vsi trace function and compared to the stored trace mask that was previously set by the user, see vsi_o_tracemask(). In this trace mask, all bits of the trace classes to be traced are set. After this comparison, the trace is either executed or not.

4.5.8.1 vsi_o_ttrace () - Trace Text

Function definition:

int vsi_o_ttrace (T_HANDLE caller, ULONG tclass, char * format, ...)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
ULONG	tclass	class to be traced	IN
char *	format	format of string to be built	IN

Return:

Type	Meaning	
int	VSI_ERROR	trace class not enabled in trace mask
	VSI_OK	success

Options: none

Description:

The function vsi_o_ttrace() compares the specified trace class to the stored trace mask for the calling task. If the bit representing the class to be traced is set in the trace mask, the trace is executed, otherwise the function returns VSI_ERROR.

This function is used to trace dynamic strings that cannot be coded to indices and decoded in the test system (see vsi_o_itrace()).

4.5.8.2 vsi_o_func_ttrace () - Trace Function Name

Function definition:

int vsi_o_func_ttrace (const char * const format, ...)

Parameters:

Type	Name	Meaning	
const char * const	format	describing the variable arguments in the list	IN
…		variable argument list	IN

Return:

Type	Meaning	
int	VSI_ERROR	trace class TC_FUNC not enabled in trace mask
	VSI_OK	success

Options: none

Description:

The function vsi_o_func_ttrace() checks if the trace class TC_FUNC is enabled for the calling entity. If yes the trace is executed, otherwise the trace is aborted and the function returns VSI_ERROR.

The handle of the calling entity is automatically added to the trace inside vsi_o_func_ttrace(). If this is a non-Task thread or a task not based on the GPF frame, the handle of the calling entity is set to 0 and the trace destination will be displayed as SYST in PCO.

It is recommended to call this trace functionality only via the TRACE_FUNCTION/TRACE_FUNCTION_P1…9 macro (4.3.39), be able to compile no-trace versions and to allow compressed tracing, refer to 06-03-42-UDO-0001.

4.5.8.3 vsi_o_event_ttrace () - Trace Event

Function definition:

int vsi_o_event_ttrace (const char * const format, ...)

Parameters:

Type	Name	Meaning	
const char * const	format	describing the variable arguments in the list	IN
…		variable argument list	IN

Return:

Type	Meaning	
int mask	VSI_ERROR	trace class TC_EVENT not enabled in trace
	VSI_OK	success

Options: none

Description:

The function vsi_o_event_ttrace() checks if the trace class TC_EVENT is enabled for the calling entity. If yes the trace is executed, otherwise the trace is aborted and the function returns VSI_ERROR.

The handle of the calling entity is automatically added to the trace inside vsi_o_event_ttrace(). If this is a non-Task thread or a task not based on the GPF frame, the handle of the calling entity is set to 0 and the trace destination will be displayed as SYST in PCO.

It is recommended to call this trace functionality only via the TRACE_EVENT/TRACE_EVENT_P1…9 macro (4.3.40), be able to compile no-trace versions and to allow compressed tracing, refer to 06-03-42-UDO-0001.

4.5.8.4 vsi_o_error_ttrace () - Trace Error

Function definition:

int vsi_o_error_ttrace (const char * const format, ...)

Parameters:

Type	Name	Meaning	
const char * const	format	describing the variable arguments in the list	IN
…		variable argument list	IN

Return:

Type	Meaning	
int mask	VSI_ERROR	trace class TC_ERROR not enabled in trace
	VSI_OK	success

Options: none

Description:

The function vsi_o_error_ttrace() checks if the trace class TC_ERROR is enabled for the calling entity. If yes the trace is executed, otherwise the trace is aborted and the function returns VSI_ERROR.

The handle of the calling entity is automatically added to the trace inside vsi_o_error_ttrace(). If this is a non-Task thread or a task not based on the GPF frame, the handle of the calling entity is set to 0 and the trace destination will be displayed as SYST in PCO.

It is recommended to call this trace functionality only via the TRACE_ERROR macro (4.3.43), be able to compile no-trace versions and to allow compressed tracing, refer to 06-03-42-UDO-0001.

4.5.8.5 vsi_o_state_ttrace () - Trace State

Function definition:

int vsi_o_state_ttrace (const char * const format, ...)

Parameters:

Type	Name	Meaning	
const char * const	format	describing the variable arguments in the list	IN
…		variable argument list	IN

Return:

Type	Meaning	
int mask	VSI_ERROR	trace class TC_STATE not enabled in trace
	VSI_OK	success

Options: none

Description:

The function vsi_o_state_ttrace() checks if the trace class TC_STATE is enabled for the calling entity. If yes the trace is executed, otherwise the trace is aborted and the function returns VSI_ERROR.

The handle of the calling entity is automatically added to the trace inside vsi_o_state_ttrace(). If this is a non-Task thread or a task not based on the GPF frame, the handle of the calling entity is set to 0 and the trace destination will be displayed as SYST in PCO.

This function is called in the macros GET_STATE() and SET_STATE() and does not need to be called ‘by hand’, State traces are processed by the compressed tracing, refer to 06-03-42-UDO-0001.

4.5.8.6 vsi_o_class_ttrace () - Trace User Trace Class

Function definition:

int vsi_o_class_ttrace (ULONG traceclass, const char * const format, ...)

Parameters:

Type	Name	Meaning	
ULONG	traceclass	class of the string to be traced	IN
const char * const	format	describing the variable arguments in the list	IN
…		variable argument list	IN

Return:

Type	Meaning	
int mask	VSI_ERROR	trace class TC_USERx not enabled in trace
	VSI_OK	success

Options: none

Description:

The function vsi_o_class_ttrace() checks if the trace class passed in the parameter *traceclass* is enabled for the calling entity. If yes the trace is executed, otherwise the trace is aborted and the function returns VSI_ERROR.

The handle of the calling entity is automatically added to the trace inside vsi_o_class_ttrace(). If this is a non-Task thread or a task not based on the GPF frame, the handle of the calling entity is set to 0 and the trace destination will be displayed as SYST in PCO.

It is recommended to call this trace functionality only via the TRACE_USER_CLASS/TRACE_USER_CLASS_P1…9 macro (4.3.41), be able to compile no-trace versions and to allow compressed tracing, refer to 06-03-42-UDO-0001.

4.5.8.7 vsi_o_ptrace () - Trace Primitive

Function definition:

int vsi_o_ptrace (T_HANDLE caller, USHORT opc, USHORT dir)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
ULONG	opc	operation code of the primitive to be traced	IN
USHORT	dir	direction of the primitive (receive/send)	IN

Return:

Type	Meaning	
int	VSI_ERROR	trace class TC_PRIM not enabled in trace mask
	VSI_OK	success

Options: none

Description:

The function vsi_o_ptrace() compares the trace class TC_PRIM to the stored trace mask for the calling task. If the bit representing the class to be traced is set in the trace mask, the trace is executed, otherwise the function returns VSI_ERROR.

This function is used to trace an operation code and the direction of a primitive. An external test system decodes the operation code and displays the primitive name and direction.

This function does not need to be called by the application itself. It is called within the frame respectively in the corresponding macro when sending/receiving primitives.

4.5.8.8 vsi_o_strace () - Trace State

Function definition:

int vsi_o_strace (T_HANDLE caller, char *machine, char *curstate, char *newstate)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
char *	machine	name of state machine	IN
char *	curstate	current state of state machine	IN
char *	newstate	new state of state machine	IN

Return:

Type	Meaning	
int mask	VSI_ERROR	trace class TC_STATE not enabled in trace
	VSI_OK	success

Options: none

Description:

The function vsi_o_strace() compares the trace class TC_STATE to the stored trace mask for the calling task. If the bit representing the class to be traced is set in the trace mask, the trace is executed, otherwise the function returns VSI_ERROR.

This function is used to trace a state or a state transition for the state machines that are defined within the protocol entities.

This function does not need to be called by the application itself. It is called within macros when setting/getting a state of a state machine.

4.5.8.9 vsi_o_primsend () - Send Primitive to PC

Function definition:

int vsi_o_primsend (T_HANDLE caller, unsigned int mask, T_HANDLE dst, char *ext_dst, unsigned int prim_id, void *ptr, unsigned int len)

If partition pool monitoring activated:

int vsi_o_primsend (T_HANDLE caller, unsigned int mask, T_HANDLE dst, char *ext_dst, unsigned int prim_id, void *ptr, unsigned int len, const char *file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
unsigned int	mask	trace filter class	IN
T_HANDLE	dst	destination entity	IN
char *	ext_dest	name of destination on PC	IN
int	prim_id	primitive id	IN
void *	ptr	pointer to primitive or data	IN
unsigned int	len	data length	IN

Return:

Type	Meaning	
int	VSI_ERROR	failed
	VSI_OK	success

Options: none

Description:

The function vsi_o_primsend() is used to send an allocated primitive identified by the parameter *ptr* or any other data identified by *ptr* and *len* to a GPF based application specified by *ext_dest* running on the connected PC.

If the *prim_id* is zero it is assumed that *ptr* points to an allocated primitive. If the *prim_id* is different from zero, the frame allocates a memory partition for a primitive, copies the data identified by *ptr* and *len* into this primitive and sends it to the PC.

The function vsi_o_primsend() should not be called directly but via on of the macros in section 4.3 like PSEND_TO_PC, refer to 4.3.34.

4.5.8.10 vsi_o_itrace () - Trace Index

Function definition:

int vsi_o_itrace (T_HANDLE caller, ULONG tclass, USHORT index, char *format, ...)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
ULONG	tclass	class to be traced	IN
USHORT	index	index to be traced	IN
char *	format	describing the variable arguments in the list	IN
…		variable argument list	IN

Return:

Type	Meaning	
int	VSI_ERROR	trace class not enabled in trace mask
	VSI_OK	success

Options: none

Description:

The function vsi_o_itrace() compares the specified trace class to the stored trace mask for the calling entity. If the bit representing the class to be traced is set in the trace mask, the trace is executed, otherwise the function returns VSI_ERROR.

This function is used to trace indices representing strings. It is a replacement for the function vsi_o_ttrace().
Instead of the string only an index is traced. The replacement of the function calls is done by an external tool that searches through the source files. This is done to save program memory and to reduce the system load as a result of the interrupts of the serial interface when transmitting long strings.

This function takes a variable number of arguments. Within the format string the information is found how many more arguments follow and of what C type they are. The format string is build out of the following characters: **c** (char), **d** (double) ,**i** (long integer) ,**p** (pointer) ,**s** (C string), * (long integer).

Every argument in the variable argument list is traced in its binary representation e.g. a long integer is traced as four successive bytes. For every character found in the format string one argument is expected in the variable argument list. The character defines the C type of the argument.

Index and arguments are transmitted in little endian byte order.

An external test system decodes the indices and displays the assigned strings.

This function MUST NOT be used directly in the implementation. The distribution of the trace indices is under exclusive control of an external tool and would otherwise be messed up.

4.5.8.11 vsi_o_func_itrace () - Trace Function - Index

Function definition:

int vsi_o_func_itrace (USHORT index, char *format, ...)

Parameters:

Type	Name	Meaning	
USHORT	index	index to be traced	IN
char *	format	describing the variable arguments in the list	IN
…		variable argument list	IN

Return:

Type	Meaning	
int	VSI_ERROR	trace class TC_FUNC not enabled in trace mask
	VSI_OK	success

Options: none

Description:

The function vsi_o_func_itrace() checks if the trace class TC_FUNC is enabled in trace mask for the calling entity. If TC_FUNC is set in the trace mask, the trace is executed, otherwise the function returns VSI_ERROR.

This function is used to trace indices representing strings. It is a replacement for the function vsi_o_func_ttrace() called in the macros TRACE_FUNCTION/TRACE_FUNCTION_P1…9 (4.3.39) in case of compressed tracing enabled.

Instead of the string only an index is traced. The replacement of the function calls is done by an external tool that searches through the source files. This is done to save program memory and to reduce the system load as a result of the interrupts of the serial interface when transmitting long strings.

This function takes a variable number of arguments. Within the format string the information is found how many more arguments follow and of what C type they are. The format string is build out of the following characters: **c** (char), **d** (double) ,**i** (long integer) ,**p** (pointer) ,**s** (C string), * (long integer).

Every argument in the variable argument list is traced in its binary representation e.g. a long integer is traced as four successive bytes. For every character found in the format string one argument is expected in the variable argument list. The character defines the C type of the argument.

The handle of the calling entity is automatically added to the trace inside vsi_o_class_ttrace(). If this is a non-Task thread or a task not based on the GPF frame, the handle of the calling entity is set to 0 and the trace destination will be displayed as SYST in PCO.

Index and arguments are transmitted in little endian byte order.

An external test system decodes the indices and displays the assigned strings.

This function MUST NOT be used directly in the implementation. The distribution of the trace indices is under exclusive control of an external tool and would otherwise be messed up.
TRACE_FUNCTION/TRACE_FUNCTION_P1…9 (4.3.39) has to used instead.

4.5.8.12 vsi_o_event_itrace () - Trace Event - Index

Function definition:

int vsi_o_event_itrace (USHORT index, char *format, ...)

Parameters:

Type	Name	Meaning	
USHORT	index	index to be traced	IN
char *	format	describing the variable arguments in the list	IN
…		variable argument list	IN

Return:

Type	Meaning	
int mask	VSI_ERROR	trace class TC_EVENT not enabled in trace
	VSI_OK	success

Options: none

Description:

The function vsi_o_event_itrace() checks if the trace class TC_EVENT is enabled in the stored trace mask for the calling entity. If TC_EVENT is set in the trace mask, the trace is executed, otherwise the function returns VSI_ERROR.

This function is used to trace indices representing strings. It is a replacement for the function vsi_o_event_ttrace() called in the macros TRACE_EVENT/TRACE_EVENT_P1…9 (4.3.40) in case of compressed tracing enabled.

Instead of the string only an index is traced. The replacement of the function calls is done by an external tool that searches through the source files. This is done to save program memory and to reduce the system load as a result of the interrupts of the serial interface when transmitting long strings.

This function takes a variable number of arguments. Within the format string the information is found how many more arguments follow and of what C type they are. The format string is build out of the following characters: **c** (char), **d** (double) ,**i** (long integer) ,**p** (pointer) ,**s** (C string), * (long integer).

Every argument in the variable argument list is traced in its binary representation e.g. a long integer is traced as four successive bytes. For every character found in the format string one argument is expected in the variable argument list. The character defines the C type of the argument.

The handle of the calling entity is automatically added to the trace inside vsi_o_class_ttrace(). If this is a non-Task thread or a task not based on the GPF frame, the handle of the calling entity is set to 0 and the trace destination will be displayed as SYST in PCO.

Index and arguments are transmitted in little endian byte order.

An external test system decodes the indices and displays the assigned strings.

This function MUST NOT be used directly in the implementation. The distribution of the trace indices is under exclusive control of an external tool and would otherwise be messed up.
TRACE_EVENT/TRACE_EVENT_P1…9 (4.3.40) has to used instead.

4.5.8.13 vsi_o_error_itrace () - Trace Error - Index

Function definition:

int vsi_o_error_itrace (USHORT index, char *format, ...)

Parameters:

Type	Name	Meaning	
USHORT	index	index to be traced	IN
char *	format	describing the variable arguments in the list	IN
…		variable argument list	IN

Return:

Type	Meaning	
int mask	VSI_ERROR	trace class TC_ERROR not enabled in trace
	VSI_OK	success

Options: none

Description:

The function vsi_o_error_itrace() checks if the trace class TC_ERROR is enabled in the stored trace mask for the calling entity. If TC_ERROR is set in the trace mask, the trace is executed, otherwise the function returns VSI_ERROR.

This function is used to trace indices representing strings. It is a replacement for the function vsi_o_error_ttrace() called in the macro TRACE_ERROR (4.3.43) in case of compressed tracing enabled.

Instead of the string only an index is traced. The replacement of the function calls is done by an external tool that searches through the source files. This is done to save program memory and to reduce the system load as a result of the interrupts of the serial interface when transmitting long strings.

This function takes a variable number of arguments. Within the format string the information is found how many more arguments follow and of what C type they are. The format string is build out of the following characters: **c** (char), **d** (double) ,**i** (long integer) ,**p** (pointer) ,**s** (C string), * (long integer).

Every argument in the variable argument list is traced in its binary representation e.g. a long integer is traced as four successive bytes. For every character found in the format string one argument is expected in the variable argument list. The character defines the C type of the argument.

The handle of the calling entity is automatically added to the trace inside vsi_o_class_ttrace(). If this is a non-Task thread or a task not based on the GPF frame, the handle of the calling entity is set to 0 and the trace destination will be displayed as SYST in PCO.

Index and arguments are transmitted in little endian byte order.

An external test system decodes the indices and displays the assigned strings.

This function MUST NOT be used directly in the implementation. The distribution of the trace indices is under exclusive control of an external tool and would otherwise be messed up. TRACE_ERROR (4.3.43) has to used instead.

4.5.8.14 vsi_o_state_itrace () - Trace State - Index

Function definition:

int vsi_o_state_itrace (USHORT index, char *format, ...)

Parameters:

Type	Name	Meaning	
USHORT	index	index to be traced	IN
char *	format	describing the variable arguments in the list	IN
…		variable argument list	IN

Return:

Type	Meaning	
int mask	VSI_ERROR	trace class TC_STATE not enabled in trace
	VSI_OK	success

Options: none

Description:

The function vsi_o_state_itrace() checks if the trace class TC_STATE is enabled in the stored trace mask for the calling entity. If TC_STATE is set in the trace mask, the trace is executed, otherwise the function returns VSI_ERROR.

This function is used to trace indices representing strings. It is a replacement for the function vsi_o_state_ttrace() called in the macros GET_STATE/SET_STATE in case of compressed tracing enabled.

Instead of the string only an index is traced. The replacement of the function calls is done by an external tool that searches through the source files. This is done to save program memory and to reduce the system load as a result of the interrupts of the serial interface when transmitting long strings.

This function takes a variable number of arguments. Within the format string the information is found how many more arguments follow and of what C type they are. The format string is build out of the following characters: **c** (char), **d** (double) ,**i** (long integer) ,**p** (pointer) ,**s** (C string), * (long integer).

Every argument in the variable argument list is traced in its binary representation e.g. a long integer is traced as four successive bytes. For every character found in the format string one argument is expected in the variable argument list. The character defines the C type of the argument.

The handle of the calling entity is automatically added to the trace inside vsi_o_class_ttrace(). If this is a non-Task thread or a task not based on the GPF frame, the handle of the calling entity is set to 0 and the trace destination will be displayed as SYST in PCO.

Index and arguments are transmitted in little endian byte order.

An external test system decodes the indices and displays the assigned strings.

This function MUST NOT be used directly in the implementation. The distribution of the trace indices is under exclusive control of an external tool and would otherwise be messed up. The function vsi_o_state_itrace() is called in the GET_STATE/SET_STATE macros in case the compressed tracing is enabled.

4.5.8.15 vsi_o_class_itrace () - Trace User Class - Index

Function definition:

int vsi_o_class_itrace (ULONG traceclass, USHORT index, char *format, ...)

Parameters:

Type	Name	Meaning	
ULONG	traceclass	trace class of passed trace data	IN
USHORT	index	index to be traced	IN
char *	format	describing the variable arguments in the list	IN
…		variable argument list	IN

Return:

Type	Meaning	
int mask	VSI_ERROR	trace class TC_EVENT not enabled in trace
	VSI_OK	success

Options: none

Description:

The function vsi_o_class_ttrace() checks if the trace class passed in the parameter *traceclass* is enabled for the calling entity. If yes the trace is executed, otherwise the trace is aborted and the function returns VSI_ERROR.

This function is used to trace indices representing strings. It is a replacement for the function vsi_o_class_ttrace() called in TRACE_USER_CLASS/TRACE_USER_CLASS_P1…9 (4.3.41) in case of compressed tracing enabled.

Instead of the string only an index is traced. The replacement of the function calls is done by an external tool that searches through the source files. This is done to save program memory and to reduce the system load as a result of the interrupts of the serial interface when transmitting long strings.

This function takes a variable number of arguments. Within the format string the information is found how many more arguments follow and of what C type they are. The format string is build out of the following characters: **c** (char), **d** (double) ,**i** (long integer) ,**p** (pointer) ,**s** (C string), * (long integer).

Every argument in the variable argument list is traced in its binary representation e.g. a long integer is traced as four successive bytes. For every character found in the format string one argument is expected in the variable argument list. The character defines the C type of the argument.

The handle of the calling entity is automatically added to the trace inside vsi_o_class_ttrace(). If this is a non-Task thread or a task not based on the GPF frame, the handle of the calling entity is set to 0 and the trace destination will be displayed as SYST in PCO.

Index and arguments are transmitted in little endian byte order.

An external test system decodes the indices and displays the assigned strings.

This function MUST NOT be used directly in the implementation. The distribution of the trace indices is under exclusive control of an external tool and would otherwise be messed up.
TRACE_USER_CLASS/TRACE_USER_CLASS_P1…9 (4.3.41) has to used instead.

4.5.8.16 vsi_o_settracemask () - Set Trace mask

Function definition:

int vsi_settracemask (T_HANDLE caller, T_HANDLE handle, ULONG mask)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_HANDLE	handle	handle of destination entity for trace mask	IN
ULONG	mask	trace mask	IN

Return:

Type	Meaning	
int	VSI_ERROR	invalid destination entity handle
	VSI_OK	success

Options: none

Description:

The function vsi_o_settracemask() writes the specified trace mask into the table of the trace masks located in the VSI. Each bit in the trace masks represents a trace class that is compared to the trace class given to the vsi trace functions.

This function is only called by the test interface process that receives the configuration messages via the serial interface.

4.5.8.17 vsi_o_gettracemask () - Get Trace mask

Function definition:

int vsi_o_gettracemask (T_HANDLE caller, T_HANDLE handle, ULONG * mask)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
T_HANDLE	handle	handle of entity of requested trace mask	IN
ULONG *	mask	trace mask	OUT

Return:

Type	Meaning	
int	VSI_ERROR	invalid destination entity handle
	VSI_OK	success

Options: none

Description:

The function vsi_o_gettracemask() reads the trace mask of the requested entity out of the table of the trace masks located in the VSI and writes it into the parameter mask.

This function is only called by the test interface process that receives the configuration messages via the serial interface.

4.5.8.18 vsi_o_assert() - Fatal Error Handling

Function definition:

int vsi_o_assert (T_HANDLE caller, USHORT cause, const char *file, int line, const char * const format,...)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
USHORT	error cause	unique error code	IN
const char *	file	source file name	IN
int	line	line in source file	IN
char *	format	describing the variable arguments in the list	IN
…		variable argument list	IN

Return:

Type	Meaning	
int	VSI_OK	success

Options: none

Description:

The function vsi_o_assert() is called if a fatal error has been detected. It traces the error message and calles os_SystemError() for OS layer specific error handling.

Depending on the implementation of os_SystemError() in the used OS layer calling vsi_o_assert() may result in a reset of the mobile or the suspension of the calling task.

A frame error indication primitive is sent to applications on the tools side that have registered to receive this.

It is recommended to call the function vsi_o_assert() via the TRACE_ASSERT (4.3.44) macro.

4.5.8.19 vsi_non_gsp_trace_register () - Register non-GSP Entity Trace System

Function definition:

T_HANDLE vsi_non_gsp_trace_register (char * name, U32 flags)

Parameters:

Type	Name	Meaning	
char *	name	entity name	IN
U32	flags	flags for trace API usage	IN

Return:

Type	Meaning	
T_HANDLE	VSI_OK	handle to be used for tracing
	VSI_ERROR	registration failed

Options: none

Description:

The function vsi_non_gsp_trace_register() allows non-GSP entities to register at the GSP trace system to use the GSP trace API. A registered non-GSP entity is able to trace via the trace API function vsi_o_ttrace().

The parameter *flags* can be used to configure the trace system to e.g. drop traces in case no trace memory is available by setting the flags parameter to TRC_NO_SUSPEND.

4.5.8.20 vsi_set_non_gsp_trace_filter () - Set Trace Filter for non-GSP Entity

Function definition:

int vsi_set_non_gsp_trace_filter (T_HANDLE non_gsp_handle, ULONG trace_filter);

Parameters:

Type	Name	Meaning	
T_HANDLE	non_gsp_handle	handle of non-GSP entity	IN
ULONG	trace_filter	new trace filter setting	IN

Return:

Type	Meaning	
T_HANDLE	VSI_OK	success
	VSI_ERROR	error

Options: none

Description:

The function vsi_set_non_gsp_trace_filter() allows the setting of trace filters for a non-GSP entity specified by the parameter *non_gsp_handle* that was previously returned by the function vsi_non_gsp_trace_register().

Trace filter settings are described in 4.2.3.

4.5.8.21 vsi_get_non_gsp_trace_handle () - Get Trace Handle of non-GSP Entity

Function definition:

T_HANDLE vsi_get_non_gsp_trace_handle (char * name);

Parameters:

Type	Name	Meaning	
char *	name	entity name	IN

Return:

Type	Meaning	
T_HANDLE	VSI_OK	handle to be used for tracing
	VSI_ERROR	error

Options: none

Description:

The function vsi_get_non_gsp_trace_handle() returns the trace handle of a non-GSP entity specified by *name* that was previously registered at the GSP trace system with the function vsi_non_gsp_trace_register().

This function returns VSI_ERROR in case no entity with the specified name is registered.

4.5.9 Partition Supervision

4.5.9.1 vsi_ppm_new () - Supervision of Allocating a Partition

Function definition:

void vsi_ppm_new (T_HANDLE caller, void * prim, const char *file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	calling task	IN
void *	prim	pointer to partition holding primitive	IN
const char *	file	source file name	IN
int	line	line in source file	IN

Return: ---------

Options: OPTIMIZE_POOL

Description:

The function vsi_ppm_new() monitors the allocation of a partition used for sending primitives. The function updates an internal partition monitoring table. The state of a partition is set to ALLOCATED. State transitions are supervised and an error message is generated in the case of an illegal transition. File and line of the caller are stored in the monitoring table.

This function must be called after a partition is allocated with vsi_c_new() if the partition memory pool supervision capability should be used. To call the function vsi_ppm_new() the macro
VSI_PPM_NEW ((T_PRIM_HEADER*)Prim,__FILE__,__LINE__);
should be inserted. Prim is the pointer that vsi_c_new() returned.

If the macro PALLOC() is used for partition allocation, the call of vsi_ppm_new() is not necessary as the function is called within the frame.

If the Option OPTIMIZE_POOL is activated, a statistic processing that evaluates the different sized primitives compared to the size of the allocated partitions and the number of allocated partitions compared to the number of available partitions is also enabled.

4.5.9.2 vsi_ppm_rec () - Supervision of Receiving a Partition

Function definition:

void vsi_ppm_rec (T_HANDLE caller, void * prim, const char *file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	calling task	IN
void *	prim	pointer to partition holding primitive	IN
const char *	file	source file name	IN
int	line	line in source file	IN

Return: ---------

Options: None

Description:

The function vsi_ppm_rec() monitors the receiving of a primitive stored in a partition. The function updates an internal partition monitoring table. The state of a partition is set to RECEIVED. State transitions are supervised and an error message is generated in the case of an illegal transition. File and line of the caller are stored in the monitoring table.

It is not needed to call vsi_ppm_rec() in pei_primitive() of an entity. It is called inside the frame when a primitive has been received.

4.5.9.3 vsi_ppm_send () - Supervision of Sending a Partition

Function definition:

void vsi_ppm_send (T_HANDLE caller, T_HANDLE rcv, void * prim, const char *file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	calling task	IN
T_HANDLE	rcv	queue handle of destination task	IN
void *	prim	pointer to partition holding primitive	IN
const char *	file	source file name	IN
int	line	line in source file	IN

Return: ---------

Options: OPTIMIZE_POOL

Description:

The function vsi_ppm_send() monitors the sending of a primitive stored in a partition. The function updates an internal partition monitoring table. The state of a partition is set to SENT. State transitions are supervised and an error message is generated in the case of an illegal transition. File and line of the caller are stored in the monitoring table.

This function is called in the function vsi_c_send() and must not be called from any protocol stack entity.

If the Option OPTIMIZE_POOL is activated, a statistic processing that evaluates the different sized primitives compared to the size of the allocated partitions and the number of allocated partitions compared to the number of available partitions is also enabled.

4.5.9.4 vsi_ppm_reuse () - Supervision of Reusing a Partition

Function definition:

void vsi_ppm_reuse (T_HANDLE caller, void * prim, const char *file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	calling task	IN
void *	prim	pointer to partition holding primitive	IN
const char *	file	source file name	IN
int	line	line in source file	IN

Return: ---------

Options: OPTIMIZE_POOL

Description:

The function vsi_ppm_reuse() monitors the reusing of a primitive stored in a partition. A reuse is done when a protocol stack entity uses a the partition of a received primitive to send the next primitive. The function updates an internal partition monitoring table. The state of a partition is set to REUSED. State transitions are supervised and an error message is generated in the case of an illegal transition. If the new primitive to be sent does not fit into the reused partition, an error message is generated. File and line of the caller are stored in the monitoring table.

If the Option OPTIMIZE_POOL is activated, there is also enabled a statistic processing that evaluates the different sized primitives compared to the size of the allocated partitions and the number of allocated partitions compared to the number of available partitions.

It is not needed to call vsi_ppm_reuse() in an entity. It is called inside the frame when a PREUSE_… macro is called.

4.5.9.5 vsi_ppm_access () - Supervision of Access of a Partition

Function definition:

void vsi_ppm_access (T_HANDLE caller, void * prim, const char *file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	calling task	IN
void *	prim	pointer to partition holding primitive	IN
const char *	file	source file name	IN
int	line	line in source file	IN

Return: ---------

Options: MEMORY_SUPERVISION

Description:

The function vsi_ppm_access() monitors the access of a primitive stored in a partition.

It is not needed to call vsi_ppm_access() in an entity. It is called inside the frame when the macro PACCESS is called.

4.5.9.6 vsi_ppm_free () - Supervision of Deallocating a Partition

Function definition:

void vsi_ppm_free (T_HANDLE caller, void * prim, const char *file, int line)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	calling task	IN
void *	prim	pointer to partition holding primitive	IN
const char *	file	source file name	IN
int	line	line in source file	IN

Return: ---------

Options: OPTIMIZE_POOL

Description:

The function vsi_ppm_free() monitors the sending of a primitive stored in a partition. The function updates an internal partition monitoring table. The state of a partition is set to FREED. State transitions are supervised and an error message is generated in the case of an illegal transition. File and line of the caller are stored in the monitoring table.

This function is called in the function vsi_c_free() and must not be called from any protocol stack entity.

If the Option OPTIMIZE_POOL is activated, a statistic processing that evaluates the different sized primitives compared to the size of the allocated partitions and the number of allocated partitions compared to the number of available partitions is also enabled.

4.5.10 Miscellaneous

These functions are only used by the test interface entity for configuration and test purposes.

4.5.10.1 vsi_object_info () - Object Information

Function definition:

int vsi_object_info (T_HANDLE caller, USHORT id, T_HANDLE handle, char * buffer, USHORT size)

Parameters:

Type	Name	Meaning	
T_HANDLE	caller	handle of calling entity	IN
USHORT	id	object identifier	IN
T_HANDLE	handle	object handle	IN
char *	buffer	buffer to store information	IN
USHORT	size	size of buffer provided by the caller	IN

Return:

Type	Meaning	
int	VSI_ERROR	error
	VSI_OK	success

Options: None

Description:

The function vsi_object_info () may be used to get information about the 'objects' that exist in the system. Each object is defined by an object identifier (OS_OBJTASK, OS_OBJQUEUE, OS_OBJTIMER, OS_OBJSEMAPHORE, OS_OBJPARTITIONPOOL, OS_OBJMEMORYPOOL) and an object handle.

The functions returns VSI_ERROR if the identifier is unknown or the handle exceeds the number of objects that exist of this type. If the handle is lower than that maximum but no object exists for this handle, then an empty string ("") is written to the buffer.

The information and the format written to the buffer depend on the information provided by the RTOS.

Appendices

A. Acronyms

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

B. Glossary

International Mobile Telecommunication 2000 (IMT-2000/ITU-2000) Formerly referred to as FPLMTS (Future Public Land-Mobile Telephone System), this is the ITU's specification/family of standards for 3G. This initiative provides a global infrastructure through both satellite and terrestrial systems, for fixed and mobile phone users. The family of standards is a framework comprising a mix/blend of systems providing global roaming. <URL: http://www.imt-2000.org/>