Technical Documentation

GENERIC PROTOCOL STACK FRAMEWORK

GPF

FUG – FRAME USERS GUIDE

Document Number:	06-03-10-UDO-0001
Version:	0.7
Status:	Draft
Approval Authority:	
Creation Date:	2001-Mar-01 by MP
Last changed:	2005-Nov-29 by RME
File Name:	frame_users_guide.doc
ECCN:	US: 5D991 Europe: EAR99

Important Notice

Change History

Date	Changed by	Approved by	Version	Status	Notes
2001-Mar-01	MP et al.		0.1		1
2001-Sep-07	MP et al.		0.2		2
2002-Jan-04	MP		0.3		3
2002-May-20	XINTEGRA		0.4	Draft	
2003-Sep-02	MP		0.5	Draft	4
2004-Apr-19	RME		0.6	Draft	5
2005-Jan-05	MP		0.7	Draft	6

Notes:

1. Being Processed Initial
2. Document number changed
3. Dynamic primitives/memory added
4. General update
5. New config prims store/unstore tracemask in FFS
6. Slightly improved partition supervison chapter

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-ISP-0002	vsipei_api.doc	VSIPEI – Frame Body Interfaces, September 2003
06-03-10-ISP-0003	os_api.doc	OS - Operating System Interface, September 2003
06-03-42-UDO-0001	str2ind_usersguide.doc	Compressed/Binary Tracing

1.1 Abbreviations

RTOS Real Time Operating System

VSI Virtual System Interface

PEI Protocol Stack Entity Interface

SAP Service Access Point

HISR High Level Interrupt Service Routine

2 Introduction

This frame users guide was written to help developers and customers to use the frame and understand the basic concepts.

3 Documentation

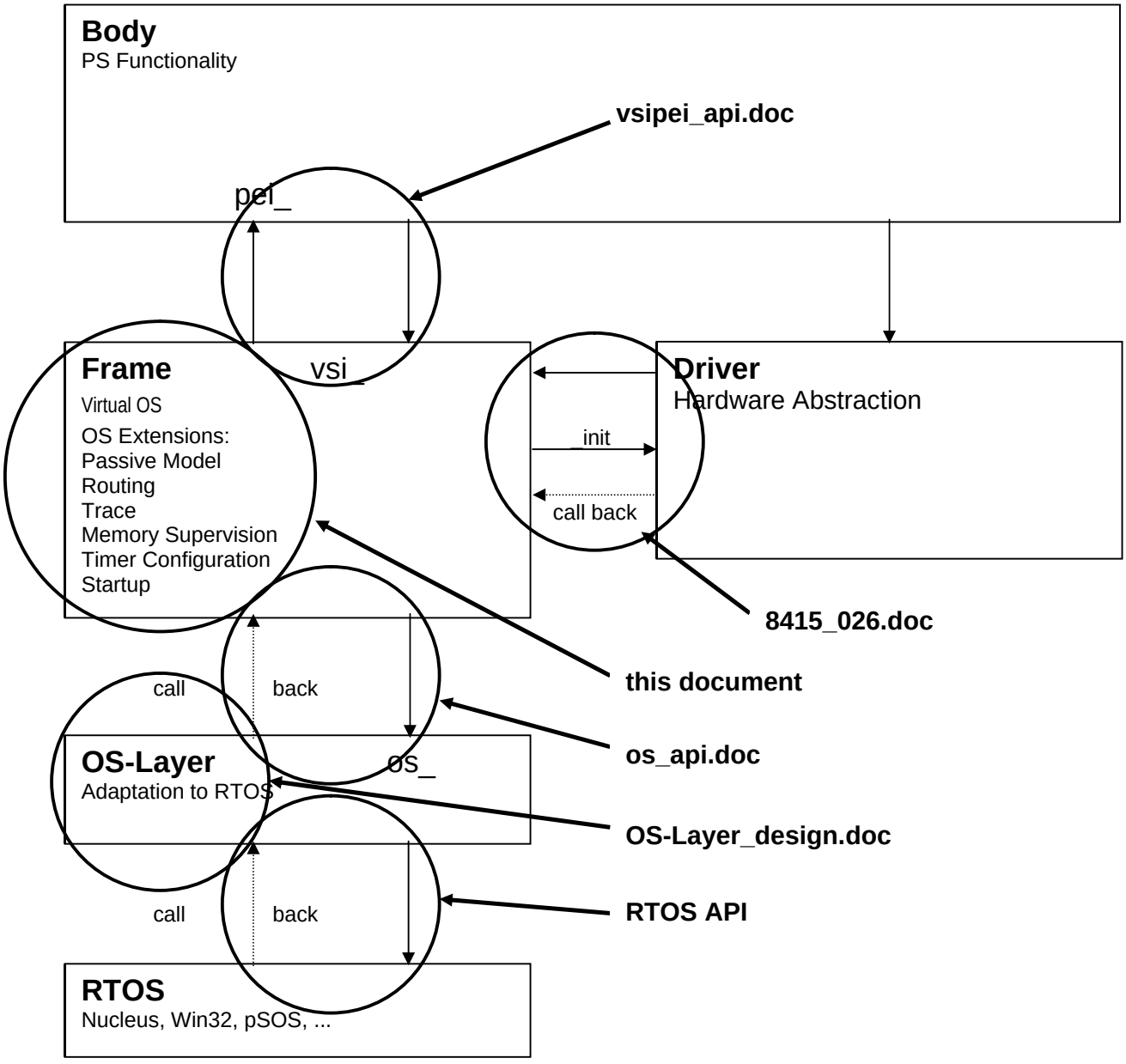


Figure 1: Protocol Stack Software Architecture and Documentation

4 Interfaces

4.1 Virtual System Interface – VSI

Refer to vsipei_api.doc

4.2 Protocol Stack Entity Interface – PEI

Refer to vsipei_api.doc

4.3 Operating System Interface – OS

Refer to os_api.doc

5 Task Management

The protocol stack entities may run as a separate operating system task or as a group of entities sharing the same task. At system startup the task list (refer to 17.2.1) is evaluated and a task/entity is created for each entry with the parameters exported by the xxx_pei_create() function. After creation all the tasks are started. All tasks are created at system startup and never deleted. The priorities are assigned at system start and are never changed during runtime. The scheduling is done by the RTOS. The feature of preemptive multitasking is used. This means that

1) a task switch will be done by the RTOS if a running task with low priority sets a task with higher priority in the READY state by writing to its queue or releasing a semaphore for which the high priority task is waiting.
2) a task switch can occur at any time due to an interrupt service routine in which a task with higher priority than the interrupted task is set in the READY state.

Some examples follow to clarify some common scenarios.

Example 1: Task A has a higher priority than Task B

	Task A state	Task B state
Task A processes a primitive	EXECUTE	BLOCKED(queue wait)
Task A write to task B's queue	EXECUTE	READY
Task A continues processing	EXECUTE	READY
Task A waits for next primitive	BLOCKED	EXECUTE
Task B processes primitive	BLOCKED	EXECUTE
Task B waits for next primitive	BLOCKED	BLOCKED

In this example the task switch is done when task is blocked.

Example 2: Task A has a lower priority than Task B

	Task A state	Task B state
Task A processes a primitive	EXECUTE	BLOCKED(queue wait)
Task A write to task B's queue	READY	EXECUTE
Task B processes primitive	READY	EXECUTE
Task B waits for next primitive	EXECUTE	BLOCKED

Task A continues processing	EXECUTE	BLOCKED
Task A waits for next primitive	BLOCKED	BLOCKED

In this example the task switch is done immediately when task B gets READY.

Example 3: Task A has a lower priority than Task B and Interrupt occurs

	Task A state	Task B state	HISR state
Task A processes a primitive	EXECUTE	BLOCKED(queue wait)	BLOCKED
Interrupt occurs, activate HISR	READY	BLOCKED	EXECUTE
HISR write to task B's queue	READY	READY	EXECUTE
HISR continues	READY	READY	EXECUTE
HISR done	READY	EXECUTE	BLOCKED
Task B processes primitive	READY	EXECUTE	BLOCKED
Task B waits for next primitive	EXECUTE	BLOCKED	BLOCKED
Task A continues processing	EXECUTE	BLOCKED	BLOCKED
Task A waits for next primitive	BLOCKED	BLOCKED	BLOCKED

This example shows how a task switch is done when a low priority task is interrupted and during the HISR a high priority task is set in the READY state.

6 Memory Management

The frame controls different types of memory pools.

6.1 Dynamic Memory

There are two dynamic memory pools that are used to allocate the task stacks during task creation and to allocate the queue memory during queue creation. These memory pools can be located in different memory sections e.g. internal RAM and external RAM. An entity can request via the flags exported by pei_create() from which of these pools its stack and queue shall be allocated.

The size of this dynamic memory pools is determined by the entries INT_DATA_POOL_SIZE and EXT_DATA_POOL_SIZE in the configuration file xxxcomp.c, refer to 17.2.1.2. In order to avoid fragmentation of the memory pool it is use only for the allocation of memory that is never freed again.

To allocate and deallocate memory from the dynamic memory pools there are currently only OS adaptation layer functions are available. There are no VSI API functions available.

Additional dynamic memory pools can be created via the configuration file xxxcomp.c or the OS layer API.

6.2 Partition Memory

In addition to the dynamic memory pool there are three groups of partition memory pools. These pools contain a number of fixed sized buffers. The size and the number of the buffers are determined in the configuration file xxxcomp.c, refer to 17.2.1.2. Each group may contain an unlimited number of partition pools. The advantage of these pools is that fragmentation will be no problem and the access is deterministic and faster than for the dynamic memory pool.

The partitions of one group of these pools are used for primitive communication and the constants for there dimensions are named PRIM_PARTITION_x_SIZE and PRIMPOOL_x_PARTITIONS in xxxconst.h. The partitions of the second group are used for are used for non-communication dynamic memory. The dimensions are determined by DMEM_PARTITION_x_SIZE and DMEMPOOL_x_PARTITIONS. The partitions of the third group are used for test interface communication and are named TEST_PARTITION_x_SIZE and TESTPOOL_x_PARTITIONS. The partitions

needed for the test interface communication are allocated from a separate pool because the primitive communication must not be affected if a lot of partitions are needed for tracing.

To allocate and deallocate a memory partition from a primitive partition pool the functions vsi_c_pnew() and vsi_c_pfree() that are hidden in the macros PALLOC() and PFREE() have to be used, refer to vsipei_api.doc (06-03-10-ISP-0002). The macros MALLOC() and MFREE which call vsi_m_cnew() and vsi_m_cfree() also access the primitive communication pool, because MALLOC() and MFREE() are used to handle the memory for the descriptor lists which may be part of primitives.

The non-communication partitions are also allocated with vsi_m_new() and vsi_m_free() which are hidden behind the macros DALLOC() and DFREE(). There is no VSI function available for the user to allocate a memory partition from the test interface partition pool. These allocations are done inside the frame within the functions for tracing or routing of primitives.

In addition to the allocation of a single block of fixed sized memory the frame offers functions to allocated a chain of memory. This may be used for dynamic sized primitives. In this case the total size of the primitive is unknown when the 'root' of the primitive is allocated. To allocate such a primitive root the function vsi_drpo_new() hidden in DRPO_ALLOC() is used. Further allocations in the same primitive can then be done with vsi_dp_new()/DP_ALLOC(), refer to vsipei_api.doc (06-03-10-ISP-0002). In the same way a chain of memory can be allocate for non-communication purposes with the function vsi_drp_new()/DRP_ALLOC() and additional vsi_dp_new()/DP_ALLOC() calls. To deallocate such a dynamic sized primitive/chain of dynamic memory the function vsi_free()/FREE() must be used.

When an attempt to allocate a memory partition cannot be satisfied immediately because of a lack of partitions of the requested size then the frame return a partition of the next available size and return a warning message, refer to 15.2.

If no free partition is available at calling time the calling task is suspended or – if the non-blocking API is used – a NULL pointer is returned. 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 (blocking API) then an error message is traced and an RTOS/target specific fatal error handling is performed.

Besides the allocation and deallocation functions the frame provides an interface to attach to an already allocated memory partition. To attach to a partition holding a primitive PATTACH() which hides vsi_c_pattach() is used, for a partition allocated with MALLOC() the MATTACH() has to be used. This attach procedure increments the reference counter in the header of an allocated partition. The reference counter is checked at each deallocation call and only if it is zero the partition will be freed.

During system start-up when the partition pools are created a guard pattern with the value 0xAFFEDEAD is written at the end of each memory partition. This guard pattern is checked every time a primitive is sent and every time a memory partition is freed. If the guard pattern is destroyed then an system error message is generated, refer to 15.3, the content of the corresponding partition is dumped and the system is reset.

There is a supervision mechanism for the partitions of the primitive partition pool available, refer to 18. To enable this a specific set of libraries has to be used, refer to 17.1. With this partition supervision it is possible to check, if all partitions are freed. Also some statistic information on the usage of the partition memory is generated.

7 Inter Process Communication

The communication between the different protocol stack entities is done via message queues. Each operating system task has one queue that is created when the task is scheduled the first time. When the initialization of all entities running in a task is done the task enters its main loop which means it waits for a message in its input queue.

Three different kinds of messages can be exchanged through a message queue: Primitives, signals and timeouts. Primitives and signals are sent from one entity to another, timeout messages are triggered by the RTOSin case of a timeout, assembled in the timeout callback function in the frame and sent to a specific entity.

To different priorities are assigned to the different messages. Primitives and timeouts have low priority, signals have high priority. This means that a signal written into a queue that already contains some primitives will be processed first.

The message exchange with primitives is done without copying of the message data. The entity that needs to send a primitive has to allocate a memory partition by calling the function vsi_c_pnew() that is hidden in the macro PALLOC(). There are also further macros for primitive that contain an SDU, refer to vsipei_api.doc (06-03-10-ISP-0002). The primitive name has to be passed to PALLOC() and it creates a pointer of the type of the requested primitive that points to an allocated memory partition (6.2) that is big enough to store the requested primitive type. PALLOC() may also add some additional parameters to the VSI interface needed for the partition memory supervision functionality if enabled. Then data has to be stored in the primitive and it has to be sent to the destination entity by calling vsi_c_psend() that is hidden in the PSEND() macro. PSEND is called with the handle of the destination entity and the pointer to the primitive to be sent. The queue handles of all the entities with which one entity needs to exchange messages have to retrieved in the entities pei_init() function (refer vsipei_api.doc, 06-03-10-ISP-0002) that is called during startup, refer to (10)13.

The frame realizes that a primitive is written into the queue of the destination entity and calls the pei_primitive() function of this entity. There the SAP number of the primitive is evaluated from the operation code of the primitive. The SAP determines a table of function addresses. Then a function of this table is called with the primitive number that is also part of the operation code as index. The received pointer to the primitive is passed to this function to process it. After processing the memory partition that was used to transport the primitive has to freed by the entity. Therefor the function vsi_c_pfree() that is hidden in the macro PFREE() has to be called.

The communication with signals is done in a similar way. The difference apart from the higher priority is that signals do not use an allocated memory partition to exchange messages. The pointer that is written into a message queue points to a static memory buffer within the sending entity. The entity that receives the signal accesses this memory buffer directly. In this case it has to ensured that the memory buffer still contains the same date when the destination entity accesses it. The advantage of signal communication is that the time for partition memory allocation is saved. For signal communication the frame calls the pei_signal() function of the destination entity when a signal is received.

For send timeout messages no memory buffer is needed. Only the index of the expired timer is put into the message queue. The frame called the entities pei_timeout() function of the entity that started this timer and passes the timer index to it.

If a message queue is full a the moment when a message should be written the writing task is suspended until there is space available in the destination queue. In this case the frame generates a system warning. This should never happen and is probably the consequence of another problem, e.g. that the destination task is suspended while it waits for a memory partition.

If the write attempt is executed during an ISR then the caller cannot be suspended until there is queue space available. This is treated as a fatal error, a system error message is traced and the system is reset. Typical ISRs that write into queues are the layer 1 and the system tick ISR that processes timeouts.

8 Timer Management

The frame provides a set of VSI functions to use application timers. A timer can be started and stopped and the time until the expiration of a timer can be requested. In addition to the timers that expire only once the frame provides periodic timers that expire periodically until they are stopped by the application.

The number of timers in the system is determined by the constant MAX_TIMER that is defined in xxxconst.h, refer to 17.2.4. MAX_SIMULTANEOUS_TIMER defines the number of timers that can run simultaneously. These two definitions have been introduced to save RAM, because of the additional management effort needed for running/expired timers.

The RTOS abstraction layer uses only a single RTOS timer to run all the application timers. Therefor it manages a list of all running timers and starts the next timer in the list with its remaining time when the

previous one has expired. This mechanism saves a lot of memory for the timer control blocks compared to an implementation where each application timer is represented by its own RTOS timer.

To access a timer from a task an index has to be passed to the corresponding VSI function. This index has to be in the range between zero and the number of timers requested by this entity in its pei_create() function.

The attempt to use more timers than available as well as the attempt to start a timer with and index bigger than the one defined in pei_create() will result in a system error message and the system to be reset.

Timers can be configured via system primitives received through the test interface. It is possible to slow down, speed up or suppress a timer. The corresponding command are described in 14.1.10.1.

9 Routing

The routing functionality is used to redirect or duplicate primitives. Redirecting a message means that it is not sent to the entity that was specified during the call of vsi_c_send() but to a different protocol stack entity or to an external test system via the test interface process and its drivers. Duplication of a primitive effects that the message is not only sent to its original destination but also to a different receiver which can be to a different protocol stack entity or to an external test system.

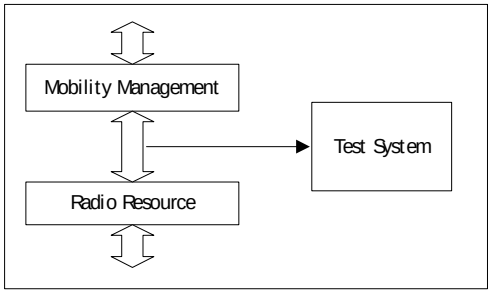
In addition to the general redirection/duplication of messages that one entity sends to another it is also possible to filter the operation code of a primitive. This allows to route specific primitives that are exchanged between to entities.

The routings are set dynamically, they are not compiled into the frame code. The user can set the routings with system primitives sent to the corresponding entity with an external program e.g. PCO. Also the test cases run with the TAP configure the protocol stack in the way it needed to run the test cases. The syntax of the system primitives to enter routings can be seen in 14.2. Routings can be entered separately for each entity.

The routing functionality can be used in four different ways:

- Observation

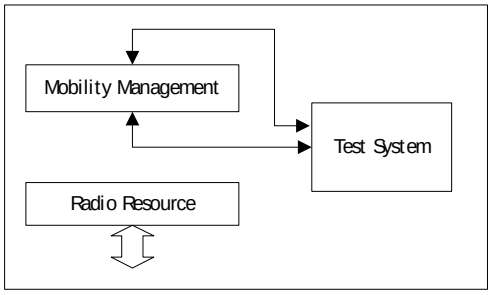
All or a certain amount of primitives exchanged between two entities are duplicated to an external observation tool like PCO. This allows the user to watch the primitive contents.



Example: Duplicate all primitive sent from RR to MM to PCO and duplicate all primitives sent from MM to RR to PCO

- Test

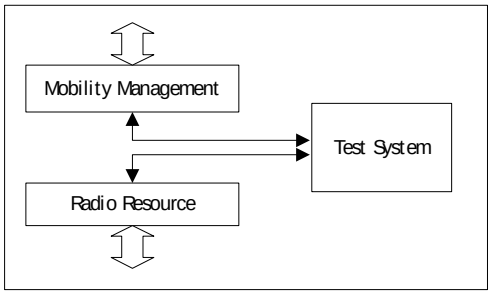
To test a single entity the communication channels to the neighbouring entities can be cut of. The entity is now isolated within the protocol stack and can be connected with an external test application like the TAP. This behaviour can be achieved with the redirection of primitives.



Example: Redirect all primitives sent from MM to the TAP and redirect all primitives that are sent from the other entities to MM to NULL which means discard them.

- Manipulation

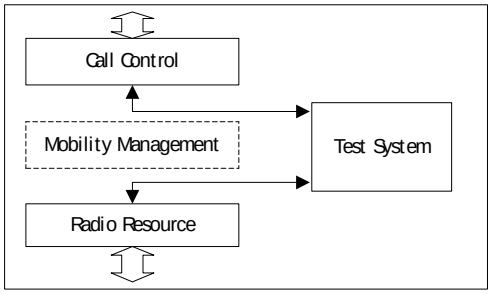
All or a certain amount of primitive exchanged between two entities are redirected to and external test system. The test system can manipulate these messages e.g. to simulate protocol errors and then forward it to the original receiver.



Example: Redirect all primitives from MM to RR to the test system manipulate them and forward them to RR.

- Simulation

A single entity within the protocol stack can be “switched off” and run on the external test system. This behaviour can be achieved with the redirection of all primitives sent to and from this entity to the test system.



Example: Redirect all primitives sent from any entity to MM to the TAP and redirect all primitives that are sent from MM to NULL which means discard them.

The problem to be solved if the protocol stack runs on the target system and the test tools run on the PC is that the alignment and data format (little/big endian) are likely to be different. In this case the primitive converter PCON has to be called in the test interface drivers to achieve platform independent message format for the test interface communication.

10 Traces

The trace functionality is used to send any information that might be useful to follow the behaviour of a protocol stack. The information to be traced is sent via the test interface and its drivers to an external observation tool like PCO.

10.1 Trace API

The frame provides different VSI functions that can be called to trace different information, refer to vsipei_api.doc (06-03-10-ISP-0002). These functions are hidden in macros so that the calls of trace functions can be removed with a certain make option at compile time.

In addition different trace classes are used to enable or disable certain traces dynamically during runtime. This can be done with system primitives sent by an external tool like PCO. The trace classes can be adjusted separately for each entity. As default only error traces (TC_ERROR) are enabled.

The following trace classes are provided by the frame:

TC_FUNC	(0x00000001)	is used in the macro TRACE_FUNCTION
TC_EVENT	(0x00000002)	is used in the macro TRACE_EVENT
TC_PRIM	(0x00000004)	is used in the macros PTRACE_IN and PTRACE_OUT
TC_STATE	(0x00000008)	is used in the macros SET_STATE and GET_STATE
TC_SYSTEM	(0x00000010)	is used for frame traces
TC_ISIG	(0x00000020)	is used to trace entity internal signals
TC_ERROR	(0x00000040)	is used in the macro TRACE_ERROR
TC_CCD	(0x00000080)	is used for CCD traces
TC_TIMER	(0x00000100)	is used inside the timer API
TC_USER1	(0x00010000)	general purpose, to be used in TRACE_USER_CLASS
…		
TC_USER8	(0x00800000)	general purpose, to be used in TRACE_USER_CLASS

The trace class is stored as 32bit value and the user can add customer defined trace classes. It has to be ensured that these customer defined classes do not conflict with the predefined trace classes mentioned above.

The following macros are provided by the frame:

- TRACE_FUNCTION

This macro is used to trace the names of function that are called during runtime. Nearly all of the functions in the layer 2/3 entities call TRACE_FUNCTION when they are entered. To enable function traces dynamically TC_FUNC has to be set for the corresponding entity.

- TRACE_ERROR

This macro is used to trace any abnormal behaviour within the entities. To enable error traces dynamically TC_ERROR has to be set for the corresponding entity.

- PTRACE_IN and PTRACE_OUT

These macros are used to trace the names of primitives that are sent or received by the protocol stack entities. PTRACE_IN has to be called in the pei_primitive function of each entity. PTRACE_OUT is called in the PSEND macro and the user does not need to care about it. To enable primitive traces dynamically TC_PRIM has to be set for the corresponding entity.

- TRACE_EVENT(_P1…P9)

This macro is used to trace any information within the entities that does not match with the macros mentioned before. To enable event traces dynamically TC_EVENT has to be set for the corresponding entity. The trace event macros with the extensions P1 to P9 can be used to trace a formatted list of parameters. A format string is similar printf can be used.

Additionally it is possible to trace the states of the entity internal state machines. This is done within the macros to set a new state (SET_STATE) or to get a state (GET_STATE).

10.2 Compressed Trace

In order to reduce the amount of code due to the presence of the constant strings to be traced and also to reduce the number of characters sent via the test interface a compressed trace mode has been introduced. A kind of pre-processor parses the source code, replaces strings to be traced with an in-

dex, assigns this index to the original string in a table and replaces the original trace API function with a function to handle the index instead of a string. On the PCO side the index for a received trace is converted into a string with the help of the table written during previous pre processing and the original string is displayed. Further information can be found in [06-03-42-UDO-0001].

11 Test Interface

The test interface is needed to be able to communicate with external test tools. It is implemented as two tasks (one for sending and one for receiving) to avoid deadlock situations. The test interface entities also have a PEI interface similar to the other protocol stack entities. The priority of the test interface tasks is the lowest in the system. Its queue size is defined in the project dependent xxxconst.h.

The test interface is needed to

- forward traces to the external observation tool (PCO)
- forward routed protocol primitives to the external test application (TAP) or PCO
- receive system/configuration messages from the PCO or TAP
- receive protocol primitives from the panel or TAP

The test interface uses a set of drivers of different layers. The layer three driver is responsible for the coding of sender, receiver, length and timestamp. The layer two driver add some error checking information and the layer one driver determines the physical resource to be used, e.g. socket or RS232.

The drivers to be used have to be entered into a driver table. This driver table is located in the configuration file xxxdrv.c and therefore under the control of the frame based application. Drivers can simply be exchanged by a modification of this table. A recompilation of the frame is not needed. Also a dynamic reconfiguration of the driver table via configuration primitives is supported.

A synchronization mechanism between protocol stack and test tools is provided. An application like the TAP can call a dedicated VSI function vsi_c_sync() to request this synchronisation. vsi_c_sync() asks the TST entity in the same environment to request the task states of the frame based tasks in the protocol stack. When all these tasks have completely started, this vsi_c_sync() returns successful. As long not all tasks in the protocol stack have completely started, the synchronization attempts are performed until a timeout has occurred. In this case vsi_c_sync() returns with an error code.

12 RTOS Adaptation Layer

The RTOS adaptation layer is needed to keep the frame itself operating system independent. The interface is specified in os_api.doc (06-03-10-ISP-0003). In the OS Layer, the request of system resources by the protocol stack entities via the VSI is adapted to the used RTOS. If an RTOS does not provide all the features that are specified in the interface description e.g. the feature of periodic timers, this must be implemented within the OS adaptation layer.

13 System Startup

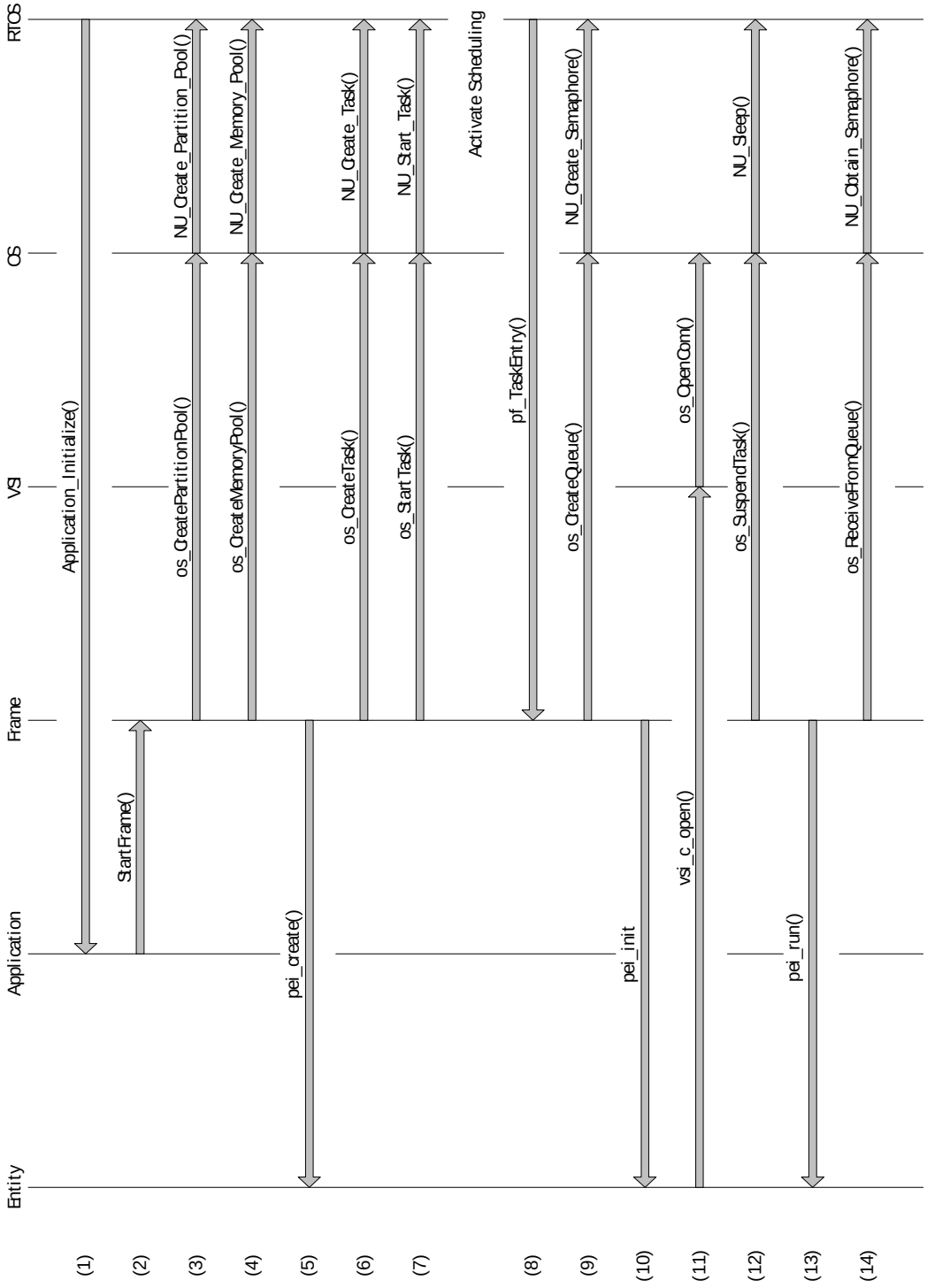


Figure xx System Startup

(1) When Nucleus has performed its internal startup procedures consisting of a target dependent and a target independent part it calls the function Application_Initialize() which is the entry point for every application running on Nucleus.

(2) Within Application_Initialize() the function StartFrame() has to be called. This is the main entry point to the frame. StartFrame() is responsible for the creation of memory pools, the creation of tasks and the entities within the tasks and the start of the tasks that are part of the current application.

(3) The partition pools needed for communication are created by calling os_CreatePartitionPool(). The number and size of the different partitions are taken from the customer controlled configuration file xxxconst.h, refer to 17.2.1.

(4) As well the memory pool to allocate task stacks and queue memory is created with os_CreateMemoryPool(). Its size is also adjusted in xxxconst.h, refer to 17.2.1.

(5) In xxxcomp.c there is a table containing the pei_create() function addresses of all the entities to run in the application. Entities may run in a single task or share a task. The frame calls each pei_create() function and creates an entity with the parameters exported by this function.

(6) A task is created by calling os_CreateTask().

(7) When all tasks are created then the frame starts all of them in the same order as they were created before. After further initialization of e.g. the L1 and the drivers Application_Initialize() returns and now the scheduling is activated in Nucleus.

(8) The tasks are scheduled in the order of there priorities with the highest priority task first. There is only one task entry function for all tasks. It is called once for each task. Each task has its own stack area to use and a handle is passed to pf_TaskEntry() to be used at all subsequent requests of frame functionality.

(9) In the task entry function a message queue for each task is created by calling os_CreateQueue(). The number of entries in the queue is taken from structure exported by pei_create().

(10) Then the pei_init() functions for all entities running in this task are called. The address of pei_init() is taken from the structure that was exported by pei_create().

(11) In pei_init() the communication communication channels to the different entities with which the currently scheduled entity needs to communicate are opened. This is done by calling vsi_c_open(). Opening communication channels means to ask the frame for the handles of the queues of these entities. If a handle cannot e requested because the corresponding task was not yet scheduled then pei_init() returns an error code and the active task is suspended for 100ms to enable Nucleus to schedule the lower priority tasks (12). Then pei_init() is called again. If all the needed handles have been retrieved then internal databases may by initialized in pei_init().

(12) A task may be suspended if not all the tasks to communicate with are scheduled at this time.

(13) If an entity is running in the active body variant, the the function pei_run() is called. The main loop of the entity is contained in pei_run().

(14) For tasks running in the passive body variant the main loop is entered. The task will enter os_ReceiveFromQueue() and be suspended until a message is received in its message queue.

14 System Primitives

System primitives are used to configure the Protocol Stack. System primitives are coded in a format understandable by humans (ASCII characters).

The following chapters describe the available test features for the frame and the syntax of the commands used to access them.

14.1 Common Configuration

14.1.1 RESET – Reset Entity

Syntax:

<*Entity*> RESET

Description: The RESET command is used to reset a single entity.

NOTE: resetting a single entity can cause a severe malfunction of the G23 Protocol Stack. No guarantee for correct functionality of the G23 Protocol Stack can be given in this case.

Example: command: RR RESET
response: OK

In this example, only the G23 Protocol Stack entity RR is reset.

14.1.2 MEMCHECK – Request Task Stack Information

Syntax:

1. <*Entity*> MEMCHECK

Description: The MEMCHECK command is used to request information about the task stacks of all entities of the G23 Protocol Stack. Beside stack base, stack size and the number of untouched stack bytes additional information about the tasks is delivered.

Example: command: RR MEMCHECK
response: Taskname:TST Stat:0 Count:209 Prio:1 Preempt:10 Slice:10 Stack-base:5b8868 Size:3000 Untouched:1528

Taskname:..... and so on for all tasks

14.1.3 STATUS – Request Status of Resources

Syntax:

<*Entity*> STATUS <*Resource*>

Description: The STATUS command is used to request information about the resources like tasks, queues, timers, semaphores and memory of the protocol stack. The parameter resource can be set to TASK, QUEUE, TIMER, SEMAPHORE, PARTITION, MEMORY

Examples: The following examples show the resource information available for the Nucleus Operating System.

command: RR STATUS QUEUE
response: Queuename: TST Startadr: 5c2148 Entries: 80 Used: 8

A similar response is given for every queue in the system.

command: RR STATUS TASK
response: Taskname: MMI Stat: 6 Count: 88 Prio: 210 Preempt: 10 Slice: 10 Stackbase: 5bda10 Size: 2048 Untouched: 1278

A similar response is given for every task in the system.

command: RR STATUS TIMER
response: Maximum 13 of 25 available timers running

A similar response is given for every timer in the system.

command: RR STATUS SEMAPHORE
response: Semname: UTST Count: 8 Suspend: 11 Waiting: 0

A similar response is given for every semaphore in the system.

command: RR STATUS PARTITION
response: Poolname: POOL21 Addr: 5b5490 PoolSize: 16640 PartSize: 200 Free: 78 Used: 2 Suspend:6

A similar response is given for every partition pool in the system.

command: RR STATUS MEMORY
response: Heapname: INTPOOL Addr: 5bce38 Size: 20000 Min: 52 Free: 7028 Suspend: 6

A similar response is given for every memory pool in the system.

14.1.4 MEMORY – Request PPM Information

Syntax:

<*Entity*> MEMORY

Description: The MEMORY command is used to request information about the state of the partitions used for primitive communication between the entities of the protocol stack. It is only implemented if the compiler option MEMORY_SUPERVISION is set. To get statistic information about the usage of the partition the option OPTIMIZE_POOL must also be set.

Example: command: RR MEMORY

response: [PPM]: ALL PARTITIONS FREED

[PPM]: NO OVERSIZE ERRORS

This should be the response if everything is working correctly.

If there is a partition which is not freed, the response is as follows:

POOL0(DMEM), PARTITION 0x58217c(16), OPC 0x0, ALLOCATED, XX, TIME 150, ccd_err.c(182)

If the option OPTIMIZE_POOL is present some additional information is provided:

[PPM]: POOL 0 (size 52)

[PPM]: MAXBYTE bytes pool 0: 172 bytes => 82%

[PPM]: MAXBYTE partitions pool 0: 0, 0, 1, 2, 1

[PPM]: MAXPART partitions pool 0: 5 partitions => 16%

[PPM]: MAXPART partitions pool 0: 0, 0, 5, 0, 0

[PPM]: MAXRANGE partitions pool 0: 0, 0, 5, 2, 2

[PPM]: TOTAL partitions pool 0: 0, 0, 73, 19, 8

Please find more information on how to interpret this output in 18.3.2.

14.1.5 SUSPENDTRACE – Suspend at Trace

Syntax: <*Destination*> SUSPENDTRACE <*Comnand*>

Description: With the SUSPENDTRACE command it is adjusted whether or not a trace shall be aborted if no partition is available or the test interface queue is full.

Example: RR SUSPENDTRACE YES

This example forces RR to wait for a partition to send the trace to the test interface and for space in the test interface queue for all traces.

RR SUSPENDTRACE NO

This example allows RR to abort traces if either no partition is available or the test interface queue is full. The number of aborted traces is counted and traced with the first message when tracing is possible again.

RR SUSPENDTRACE ALL YES

This example forces all entities to wait for a partition to send the trace to the test interface and for space in the test interface queue for all traces. This command can be sent to any valid destination within the stack.

RR SUSPENDTRACE ALL NO

This example allows all entities to abort traces if either no partition is available or the test interface queue is full. The number of aborted traces is counted and traced with the first message when tracing is possible again. This command can be sent to any valid destination within the stack.

14.1.6 ROUTE_DESCLIST – Route Data in Descriptor List

Syntax: <*Destination*> ROUTE_DESCLIST

Description: With the ROUTE_DESCLIST command the duplication/redirection of payload data transported in a descriptor list attached to a primitive can be activated for the selected entity. Per default only the primitive containing a pointer to the head of the descriptor list is duplicated/redirected.

Example: PPP ROUTE_DESCLIST

This example enables the routing of the complete descriptor lists for primitives sent by PPP.

14.1.7 CHECK_DESCLIST – Check Memory in Descriptor List

Syntax: <*Destination*> CHECK_DESCLIST

Description: With the command CHECK_DESCLIST the integrity check of a descriptor list attached to a primitive is enabled. When sending a primitive with descriptor list for each descriptor in the list and – if present – its separately attached data buffer it is checked if they are located in allocated partition memory and if the partition guard patterns are ok. If any of these conditions is not true, this will be treated as fatal error, refer to 15.3. Per default only the memory integrity of the root primitive is checked.

Example: PPP CHECK_DESCLIST

This example enables the integrity check of the complete descriptor lists for primitives sent by PPP.

14.1.8 READ_COM_MATRIX – Read Communication Matrix

Syntax: <*Destination*> READ_COM_MATRIX

Description: With the command READ_COM_MATRIX all opened communication channels in the protocol stack can be read. All communication channels opened via vsi_c_open() are traced.

Example: TST READ_COM_MATRIX

Response: MMI CC, MMI MM, MMI SMS, …, L1 PL, L1 GRR

14.1.9 REG_ERROR_IND – Register for Error/Warning Primitives

Syntax: <*Destination*> REG_ERROR_IND

Description: With the command REG_ERROR_IND an application running on the tool side can register to receive FRM_ERROR_IND and FRM_WARNING_IND primitives from the

frame which hosts protocol stack. This can be used by the test application program (TAP) to generate dedicated exit codes for error/warnings detected by the frame in the protocol stack.

Example: TST REG_ERROR_IND

14.1.10 CONFIG – Dynamic Configuration

14.1.10.1 Timer Configuration

14.1.10.1.1 TIMER_RESET

Syntax:

TIMER_RESET = <*TimerName*>

Description: The TIMER_RESET command is used to set the timer value of the timer <*TimerName*> to its default while the entity is running. The timer value can be changed using the TIMER_SET, TIMER_SPPED_UP, TIMER_SLOW_DOWN and TIMER_SUPPRESS.

The parameter <*TimerName*> is entity-specific. A description of the available timers that can be configured can be found in the chapters describing the entity-related timers.

Example: command: CC CONFIG TIMER_RESET T303
response: OK (CONFIG TIMER_RESET T303)

In this example, the timer T303 of the entity CC will be run with the value passed to vsi_t_start().

14.1.10.1.2 TIMER_SET

Syntax:

TIMER_SET <*TimerName*> <*Value*>

Description: The TIMER_SET command is used for dynamic configuration of the timers of the G23 Protocol Stack entities. Dynamic configuration of timers means to change the default values of the timers while the entity is running.

The entity will set the value of the timer <*TimerName*> to the given value <*Value*>. The units of the timers are milliseconds. The value is to be entered as a decimal number. Use the TIMER_RESET command to set the value of the timer to its default.

The parameter <*TimerName*> is entity-specific. A description of the available timers that can be configured can be found in the chapters describing the entity-related timers.

Example: command: CC CONFIG TIMER_SET T303 100
response: OK (CONFIG TIMER_SET T303 100)

In this example, the timer T303 of the entity CC is set to 100ms.

14.1.10.1.3 TIMER_SLOW_DOWN

Syntax:

TIMER_SLOW_DOWN <*TimerName*> <*Factor*>

Description: The TIMER_SLOW_DOWN command is used for dynamic configuration of the timers of the G23 Protocol Stack entities. Dynamic configuration of timers means to change the default values of the timers while the entity is running.

The entity will slow down its timer <*TimerName*> by the given factor <*Factor*>. This means the timer needs <*Factor*> * default time to expire. Use the TIMER_RESET command to set the value of the timer to its default.

The parameter <*TimerName*> is entity-specific. A description of the available timers that can be configured can be found in the chapters describing the entity-related timers.

Example: command: CC CONFIG TIMER_SLOW_DOWN T303 2
response: OK (CONFIG TIMER_SLOW_DOWN T303 2)

In this example, the timer T303 of the entity CC needs 100% more time to expire.

14.1.10.1.4TIMER_SPEED_UP

Syntax:

TIMER_SPEED_UP <*TimerName*> <*Factor*>

Description: The TIMER_SPEED_UP command is used for dynamic configuration of the timers of the G23 Protocol Stack entities. Dynamic configuration of timers means to change the default values of the timers while the entity is running.

The entity will speed up its timer <*TimerName*> by the given factor <*Factor*>. This means the timer needs default / <*Factor*> time to expire. Use the TIMER_RESET command to set the value of the timer to its default.

The parameter <*TimerName*> is entity-specific. A description of the available timers that can be configured can be found in the chapters describing the entity-related timers.

Example: command: CC CONFIG TIMER_SPEED_UP T303 2
response: OK (CONFIG TIMER_SPEED_UP T303 2)

In this example, the timer T303 only requires 50% of its default time to expire.

14.1.10.1.5TIMER_SUPPRESS

Syntax:

TIMER_SUPPRESS <*TimerName*>

Description: The TIMER_SUPPRESS command is used for dynamic configuration of the timers of the G23 Protocol Stack entities. Dynamic configuration of timers means to change the default values of the timers while the entity is running.

The entity will not start its timer <*TimerName*> at any time. Use the TIMER_RESET command to reset the timer to its default behavior.

The parameter <*TimerName*> is entity-specific. A description of the available timers that can be configured can be found in the chapters describing the entity-related timers.

Example: command: MM CONFIG TIMER_SUPPRESS T3210
response: OK (CONFIG TIMER_SUPPRESS T3210)

In this example, the starting of the timer T3210 is suppressed.

14.1.10.1.6TIMER_CLEAN

Syntax:

TIMER_CLEAN

Description: The TIMER_SUPPRESS command is used to clean up the dynamic timer configuration table. All stored configurations of all entities are deleted.

Example: command: MM CONFIG TIMER_CLEAN
response: OK (CONFIG TIMER_CLEAN)

14.2 SAP Configuration

14.2.1 DUPLICATE – Duplicate Primitives

Syntax:

1. <*OrgSource*> DUPLICATE <*OrgDestination*>[<*opc mask*>] <*AddDestination*>
2. <*OrgSource*> DUPLICATE ALL <*AddDestination*>
3. <*OrgSource*> DUPLICATE <*OrgDestination*>[<*opc mask*>] <*AddDestination*> [CLEAR]
4. <*OrgSource*> DUPLICATE CLEAR

Description: The DUPLICATE command is used to observe a specific SAP between two entities of the G23 Protocol Stack. All primitives of a specific SAP sent from the entity <*Org-Source*> to the entity <*OrgDestination*> are also sent to the entity <*AddDestination*>. With the optional parameter <*opc mask*> it is possible to duplicate only a primitive with an opc that matches with opc mask. Opc mask is specified as string consisting of the character '1' for a bit in the primitive opc that must be set to duplicate the primitive, '0' for a bit in the primitive opc that must not be set to duplicate the primitive and '*' for ignore.

The optional parameter CLEAR can be used to stop the observation of the SAP defined by the parameters <*OrgSource*> and <*OrgDestination*>.

In the case of Syntax 3, the routing table of <*OrgSource*> is deleted.

To observe more than one SAP, call the command for each SAP which is to be observed. To duplicate all primitives sent by one entity to a new destination e.g. the TAP the parameter <*OrgDestination*> has to be set to ALL.

For primitives containing a descriptor list to transport payload data it can be selected via the ROUTE_DESCLIST command if the complete data in the descriptor list has to be duplicated or as per default only the primitive including the pointer to the descriptor list.

Example:

```
command:    RR DUPLICATE MM PCO
response:   OK
command:    MM DUPLCATE RR PCO
response:   OK
```

In this example, the complete SAP RR is to be observed. All primitives which are sent from RR to MM and vice versa are also sent to the test system (PCO).

```
command:    RR DUPLICATE MM PCO CLEAR
response:   OK
command:    MM DUPLICATE RR PCO CLEAR
response:   OK
```

In this example, the observation of the SAP RR is cleared.

```
command:    RR DUPLICATE ALL PCO
response:   OK
```

In this example, all primitives which are sent from RR are duplicated via the test interface to PCO.

To read the stored redirections the command ROUTING is used (see 2.3).

14.2.2 REDIRECT – Redirect Primitives

Syntax:

5. <*OrgSource*> REDIRECT <*OrgDestination*>[<*opc mask*>] <*NewDestination*>
6. <*OrgSource*> REDIRECT ALL <New*Destination*>
7. <*OrgSource*> REDIRECT <*OrgDestination*>[<*opc mask*>] <*NewDestination*> [CLEAR]
8. <*OrgSource*> REDIRECT <*OrgDestination*> NULL
9. <*OrgSource*> REDIRECT CLEAR

Description: The REDIRECT command is used to observe a specific SAP between two entities of the G23 Protocol Stack. All primitives of a specific SAP sent from the entity <*OrgSource*> to the entity <*OrgDestination*> are also sent to the entity <*NewDestination*>. With the optional parameter opc mask it is possible to duplicate only a primitives with an opc that matches with opc mask. Opc mask is specified as string consisting of the character '1' for a bit in the primitive opc that must be set to duplicate the primitive, '0' for a bit in the primitive opc that must not be set to duplicate the primitive and '*' for ignore.

The optional parameter CLEAR can be used to stop the observation of the SAP defined by the parameters <*OrgSource*> and <*OrgDestination*>.

In the case of Syntax 7, the routing table of <*OrgSource*> is deleted.

To observe more than one SAP, call the command for each SAP which is to be observed. A maximum of 3 routing entries (DUPLICATE and REDIRECT) can be set per entity.

To redirect all primitives sent by one entity to a new destination e.g. the TAP the parameter <*OrgDestination*> has to be set to ALL.

For primitives containing a descriptor list to transport payload data it can be selected via the ROUTE_DESCLIST command if the complete data in the descriptor list has to be redirected or as per default only the primitive including the pointer to the descriptor list.

Example:

```
command:    RR REDIRECT MM PCO
response:   OK
command:    MM DUPLCATE RR PCO
response:   OK
```

All primitives which are sent from RR to MM and vice versa are redirected to the test system (PCO).

```
command:    RR REDIRECT MM PCO CLEAR
response:   OK
command:    MM REDIRECT RR PCO CLEAR
response:   OK
```

In this example, the observation of the SAP RR is cleared.

```
command:    RR REDIRECT MM NULL
response:   OK
```

In this example, all primitives which are sent from RR to MM are discarded.

command: RR REDIRECT ALL TAP
response: OK

In this example, all primitives which are sent from RR are redirected via the test interface to the TAP.

To read the stored redirections the command ROUTING is used (see 2.3).

14.2.3 ROUTING – Request Stored Routings

To read out the stored routings the command

command: <*Entity*> ROUTING

is used. The answer is given in the same syntax as the routings are stored.

< *Orgsource*> <*Command*> <*OrgDestination*>*[opcmask]* <*NewDestination*>

Example: command: RR ROUTING
response: RR REDIRECT MM ********00001111 TAP

In this example, all primitives from RR to MM with the bit 0...3 in the opc set to 1 and bit 4...7 set to 0 are redirected to the test system.

14.3 TRACECLASS – Enter Traceclass

Syntax:

<*Entity*> TRACECLASS <*ClassMask*>

Description: The TRACECLASS command is used to set the class of traces that are to be reported. Each entity of the G23 Protocol Stack may have a set of traces implemented that are divided into classes. The parameter <*ClassMask*> is represented as a two digit ASCII hex value. This value is a bit mask. Each bit identifies a specific trace class. The entity <*Entity*> will report traces of a specific class when the corresponding bit is set.

If the parameter <*Entity*> is set to TST (test interface) then the passed trace class is valid for all entities in the protocol stack.

The TRACECLASS command overwrites the trace class previously set for the entity. The following table contains a list of the available trace classes (Bits) and a description.

Mask	Short Name	Description
0x00000001	TC_FUNC	Enables traces output with TRACE_FUNCTION.
0x00000002	TC_EVENT	Enables traces output with TRACE_EVENT(_Px)
0x00000004	TC_PRIM	Enables traces output with PTRACE_IN/OUT.
0x00000008	TC_STATE	Enables state traces
0x00000010	TC_SYSTEM	Enables the frame traces.
0x00000020	TC_ISIG	Enables internal signals
0x00000040	TC_ERROR	Enables traces output with TRACE_ERROR
0x00000080	TC_CCD	Enables CCD traces
0x00000100	TC_TIMER	Enables timer traces (start, stop, expire)
0x00010000	TC_USER1	Enables user defined trace class
…	…	…
0x00800000	TC_USER8	…

Table1

Example: command: RR TRACECLASS 03
response: OK (RR TRACECLASS 03)

In this example, the radio resource entity will trace functions and events. Bits 0 and 1 are set.

command: TST TRACECLASS 00
response: OK (TST TRACECLASS 00)

In this example, the traces of all entities will be disabled.

command: RR TRACECLASS FFFFFFFF
response: OK (RR TRACECLASS FFFFFFFF)

In this example, the radio resource entity will send traces of all available trace classes.

To read out the adjusted trace class the command

command: <*Entity*> TRACECLASS

is used. The answer is given in the same syntax as the command to adjust the trace class.

<*Entity*> TRACECLASS FFFFFFFF

means that all trace classes are enabled.

14.4 TRACEMASK_IN_FFS – Store trace mask in FFS

Syntax:

<*Entity*> TRACEMASK_IN_FFS

Description: The TRACEMASK_IN_FFS command is used to store the current global TRACECLASS state into the flash file system. Each entity of the G23 Protocol Stack may have a set of trace classes stored in this trace mask. A specific stored state will be recovered after each target reset as long as the file /var/dbg/tracemask remains unchanged. **The only <*Entity*> reacting to this config primitive is RCV**. RCV is the receiver part of TST.

Example: command: RCV TRACEMASK_IN_FFS
response: OK (RCV CONFIG TRACEMASK_IN_FFS)

command: <*Entity*> TRACEMASK_IN_FFS

is used with RCV as the only valid entity. All other given entity names will result in a void config operation.

RCV TRACEMASK_IN_FFS

means that the trace mask array is stored into the flash file system.

14.5 NO_TRACEMASK_IN_FFS – Restore trace mask

Syntax:

<*Entity*> NO_TRACEMASK_IN_FFS

Description: The NO_TRACEMASK_IN_FFS command is used to restore the default global TRACECLASS state. The file in the flash file system storing the actual state is deleted. **The only <*Entity*> reacting to this config primitive is RCV.** RCV is the receiver part of TST.

Example: command: RCV NO_TRACEMASK_IN_FFS
response: OK (RCV CONFIG NO_TRACEMASK_IN_FFS)

command: <*Entity*> NO_TRACEMASK_IN_FFS

is used with RCV as the only valid entity. All other given entity names will result in a void config operation.

RCV NO_TRACEMASK_IN_FFS

means that the flash file system holding the stored trace mask array is deleted and the actual trace mask is reset to the system default.

15 System Messages and Error Handling

15.1 Traces

Trace: "All tasks entered main loop"

Meaning: All tasks have been created and started correctly, all queues were created, their pei_init() functions returned successfully and they finally entered their main loop.

Trace: "OK (*Command*)"

Meaning: Correct system primitive received and processed.

15.2 System Warnings

In case of an abnormal but non-critical situation the frame generates and traces a SYSTEM WARNING. Depending on the implementation of the linked OS layer these warnings may be stored in FFS. For the Nucleus OS layer coexisting with the Riviera framework the Diagnose and Recovery (DAR) API is called to do this. The available warnings are listed below:

Trace: "SYSTEM WARNING: Invalid system primitive '*string'* "

Cause: Unknown system primitive or system primitive with wrong parameters received.

To do: Check sent command.

Trace: " SYSTEM WARNING: Waited for space in queue, entity *entity*, queue *queue*, *file(line)*"

Cause: Input message queue of *entity* was temporarily full.

To do: Check queue size in *entity*_pei_create(). Probably reason is that destination entity processes its received primitives to slow.

Trace: "SYSTEM WARNING: Waited for partition *entity*, size *size*, *file(line)*"

Cause: All partitions of requested size were used.

To do: Check number of partitions in *xxxconst.h*. Probably reason is that some partitions are not freed after primitive processing.

Trace: "SYSTEM WARNING: Partition Deallocation failed in *entity, file(line)*"

Cause: Nucleus returned error in deallocation function.

To do: Check pointer passed to PFREE/MFREE

Trace: "SYSTEM WARNING: Partition already freed in *entity, file(line)*"

Cause: Attempt to free a partition that is not allocated.

To do: Check history of pointer passed to PFREE/MFREE.

Trace: "SYSTEM WARNING: Bigger partition allocated than requested, *entity, size, file(line)*"

Cause: The partition pool of the requested size is exhausted. Bigger partition is allocated.

To do: Check code for memory leaks. Maybe only the number of partitions in xxxconst.h has to be increased. This warning can indicate a short term peak load of memory allocation or a memory leak.

Trace: "SYSTEM WARNING: Allocation request truncated (*size->max_prim_size)*, *entity, file(line)*"

Cause: The requested size exceeds the maximum partition size due to the presence and value of the *guess* parameter in the dynamic primitive allocation functions.

To do: Check *guess* parameter in allocation function

Trace: "SYSTEM WARNING: Out of Memory - routing command rejected"

Cause: The dynamic memory pool the allocate the memory for the routing table is exhausted.

To do: Check memory pool size. Increase pool size or reduce number of routing commands.

Trace: "SYSTEM WARNING: Receiver process *entity* unknown""

Cause: Primitive received for unknown entity.

To do: Check test document or receiver selected in panel

Trace: "SYSTEM WARNING: Number of written > requested partition size in *file* (*line)*"

Cause: More bytes of a partition used than requested. Only if memory supervision active.

To do: Check number of requested bytes, especially in PALLOC_SDU. Check parameters of previous memset/memcpy calls.

The following warnings are only generated by the frame when compiled with partition supervision:

Trace: "SYSTEM WARNING: *entity* freed partition belonging to *entity*, *file(line)*"

Cause: An entity freed a memory partition that belongs to a different entity.

To do: Check if an entity frees a memory partition twice or after it was sent.

Trace: "SYSTEM WARNING: *entity* freed partition stored in *entity* queue, *file(line)*"

Cause: An entity freed a memory partition that was stored in a different entities queue.

To do: Check if an entity frees a memory partition after it was sent.

15.3 System Errors

In case of fatal error condition detected by the frame it generates and traces a SYSTEM ERROR. Depending on the implementation of the linked OS layer these errors may be stored in FFS and the mobile may be reset. For the Nucleus OS layer coexisting with the Riviera framework the Diagnose and Recovery (DAR) API is called to store the error messages and reset the mobile. The available errors are listed below:

Trace: "SYSTEM ERROR: *Task* Stack overflow in *file (line)*, *opc*"

Cause: Stack overflow in *task*, when sending primitive *opc.*

To do: Check code for large stack variables. Increase stack size of entities running in *task.*

Trace: "SYSTEM ERROR: Error at creating *task* queue in *file (line)*"

Cause: Unable to create message queue of started task *task*.

To do: Increase size of memory pool in *xxx*const.h.

Trace: "SYSTEM ERROR: Number of entities > MAX_ENTITIES in *file (line)* "

Cause: Number of entries in the task list in *xxx*comp.c exceeds MAX_ENTITIES.

To do: Increase MAX_ ENTITIES in xxxconst.h.

Trace: "SYSTEM ERROR: Partition Guard Pattern destroyed in dynamic primitive (PSEND), *entity*, *prim, opc, bad partition, file(line) n"*

Cause: The guard pattern between the Nucleus partitions was destroyed in a partition of a dynamic primitive. More bytes written than allocated.

To do: Check number requested bytes. Check parameters of previous memset/memcpy calls.

Trace: "SYSTEM ERROR: Partition Guard Pattern destroyed in desclist (PSEND), *entity*, *prim, opc, bad partition, file(line)"*

Cause: The guard pattern between the Nucleus partitions was destroyed in a partition of a descriptor list. More bytes written than allocated.

To do: Check number requested bytes. Check parameters of previous memset/memcpy calls.

Trace: "SYSTEM ERROR: Freed partition sent in desclist (PSEND), *entity*, *prim, opc, freed partition, file(line)"*

Cause: A freed partition was found in the descriptor list attached to a sent primitive.

To do: Check building of descriptor list.

Trace: "SYSTEM ERROR: Pointer to non-partition memory in desclist (PSEND), *entity*, *prim, opc partition, file(line)"*

Cause: Non partition memory was found in the descriptor list attached to a sent primitive.

To do: Check building of descriptor list.

Trace: "SYSTEM ERROR: Partition Guard Pattern destroyed (PSEND), *entity*, *prim, opc, file(line)"*

Cause: The guard pattern between the Nucleus partitions was destroyed. More bytes written than allocated.

To do: Check number requested bytes. Check parameters of previous memset/memcpy calls.

Trace: "SYSTEM ERROR: Partition Guard Pattern destroyed (PFREE), *entity*, *prim, opc, file(line)"*

Cause: The guard pattern between the Nucleus partitions was destroyed. More bytes written than allocated.

To do: Check number requested bytes. Check parameters of previous memset/memcpy calls.

Trace: "SYSTEM ERROR: Partition Guard Pattern destroyed (MFREE), *entity*, *prim, opc, file(line)"*

Cause: The guard pattern between the Nucleus partitions was destroyed. More bytes written than allocated.

To do: Check number requested bytes. Check parameters of previous memset/memcpy calls.

Trace: "SYSTEM ERROR: No Partition available, *entity, size, file(line)*"

Cause: Partition memory allocation failed. All partitions allocated or requested size exceeds biggest partition size.

To do: Check number of partitions in *xxx*const.h. Try to find out if all previously used partitions are freed. Check allocated partitions in pools.

Trace: "SYSTEM ERROR**:** Traced string to long in *file(line)*"

Cause: Length of string to be traced exceeds size of internal trace buffer.

To do: Use shorter string or compressed trace.

Trace: " SYSTEM ERROR: *entity* write attempt to *entity* queue failed, *file(line)*"

Cause: Input message queue of *entity* was full.

To do: Check queue size in *entity*_pei_create(). Probably reason is that *entity* processes its received primitives to slow. Check task status in system dump that is traced

Trace: "SYSTEM ERROR: Number of created semaphores > MAX_SEMAPHORES in *file(line)*""

Cause: To many semaphores created in the system.

To do: Increase MAX_SEMAPHORES in xxxconst.h

Trace: "SYSTEM ERROR: PREUSE - oversize error in *file*, *line*"

Cause: More bytes written into a reused partition than partition size.

To do: Check parameter for REUSE macro call..

Trace: "SYSTEM ERROR: Number of Timers > MAX_TIMER, *file (line)* "

Cause: Total number of Timers requested in all pei_create() functions exceeds MAX_TIMER.

To do: Increase MAX_TIMER in xxxconst.h.

Trace: "SYSTEM ERROR: TimerIndex > NumOfTimers for *entity*, *file (line)* "

Cause: A timer index >= the number of timer exported by pei_create() was passed to the timer API.

To do: Check *entity*_pei_create().

Trace: "SYSTEM ERROR: Number of started timers > MAX_SIMULTANEOUS_TIMER in *file (line)*"

Cause: To many simultaneous running timers in system.

To do: Increase MAX_SIMULTANEOUS_TIMER in xxxconst.h

Trace: "SYSTEM ERROR: OS initialization error *file (line)*"

Cause: Initialization of OS layer failed.

To do: Increase MAX_… in xxxconst.h

Trace: "SYSTEM ERROR: Error at creating *task* task *file (line)*"

Cause: Task creation failed.

To do: Check MAX_OS_TASKS… in xxxconst.h. Check size of memory pool where task stacks are allocated from.

Trace: "SYSTEM ERROR: Timeout write attempt to *entity* queue failed *file (line)*"

Cause: Writing timeout message to queue failed. Queue full, entity probably blocked or too slow in primitive processing.

To do: Check task status in system dump that is traced.

Trace: "SYSTEM ERROR: MFREE to non-partition memory, *entity*, *ptr*, *file (line)*"

Cause: A pointer to non-partition memory was passed to MFREE.

To do: Check pointer.

Trace: "SYSTEM ERROR: PFREE to non-partition memory, *entity*, *prim*, *file (line)*"

Cause: A pointer to non-partition memory was passed to PFREE.

To do: Check pointer.

Trace: "SYSTEM ERROR: FREE to non-partition memory, *entity*, *prim*, *file (line)*"

Cause: A pointer to non-partition memory was passed to PFREE.

To do: Check pointer.

Trace: "SYSTEM ERROR: MATTACH to non-partition memory, *entity*, *ptr*, *file (line)*"

Cause: A pointer to non-partition memory was passed to MATTACH.

To do: Check pointer.

Trace: "SYSTEM ERROR: MATTACH to free memory, *entity*, *ptr*, *file (line)*"

Cause: A pointer to non-partition memory was passed to MATTACH.

To do: Check pointer.

Trace: "SYSTEM ERROR: Ref Cnt Semaphore overrun, *entity*, *ptr*, *file (line)*"

Cause: The semaphore to protect the reference counter for partitions was overrun.

To do: Check contexts of MFREE/MATTACH and PFREE/PATTACH calls.

Trace: "SYSTEM ERROR: Magic number in dp_header destroyed (PSEND), entity*, opc, partition ,file (line)*"

Cause: Magic number that protects the dp_header in a dynamic primitive was destroyed.

To do: Check allocation parameters of dynamic primitive. Check memset/memcpy parameters during primitive access.

Trace: "SYSTEM ERROR: CTB: Cannot create IDLE task *,file (line)*"

Cause: IDLE task needed for common timer base cannot be created.

To do: Check MAX_OS_TASKS… in xxxconst.h. Check size of memory pool where task stacks are allocated from.

Trace: "SYSTEM ERROR: No memory available in TR driver, TR_RcvBufferSize = *size, file (line)*"

Cause: The test interface receive buffer cannot be allocated.

To do: Check size of memory pool where it is allocated from.

16 Profiler Support

The profiler support in the frame is currently limited to calling the profiler API functions when and entity is created or deleted and when an entity switch is detected. Logging of ask creation, deletion and switching is handled in the RTOS directly. In addition the VSI API (refer to vsipei_api.doc, 06-03-10-ISP-0002) provides some profiler macros to log function entry, function exit and points of interest.

To call the profiler API functions in the frame the profiler needs to register in the frame to pass the addresses of its API functions. Calls to the profiler API are skipped as long as these are not registered.

17 Project Setup

To setup an application that uses the frame some libraries and some configuration files have to be added to the project.

17.1 Libraries

Three libraries are supplied by GPF to be linked to the application.

- The frame library contains the functionality of startup, communication, timers, semaphores, tracing and routing.
- The test interface library contains the test interface and the drivers that can be used.
- The miscellaneous library contains common functions for data conversion.

Debug versions are available for these libraries. They do not only contain the compiler generated debug information but also additional functionality to evaluate problems.

The libraries can be found in the the gpf\lib directory. Three versions of each library are provided. First one does neither contain debug information nor partition memory supervision (no extension), the second one contains debug information (_db) and the third one both (_db_ps). The extension _pc indicates the usage of the primitive converter PCON. The Libraries are provided for the supported operating systems Nucleus MNT (_npc, MSDev6 compiler), Nucleus ARM7 (_na7, cl470 v1.22e) and Nucleus for ARM9 (_na9, cl470 v2.24). For the_na7 variant the presence of the Riviera frame work is assumed, for the _na9 variants _rv determines whether or not Riviera is present **). The extension _ir and _fl determine if the code is running from internal RAM (ir) or flash (fl).

The following libraries are provided in frame release 2.10.0:

frame_na7.lib	frame_na7_db.lib	frame_na7_db_ps.lib
frame_na7_ir.lib	frame_na7_db_ir.lib	frame_na7_db_ps_ir.lib
frame_na7_fl.lib	frame_na7_db_fl.lib	frame_na7_db_ps_fl.lib
frame_na9.lib	frame_na9_db.lib	frame_na9_db_ps.lib
frame_na9_rv.lib	frame_na9_db_rv.lib	frame_na9_db_ps_rv.lib
frame_npc.lib	frame_npc_db.lib	frame_npc_db_ps.lib
misc_na7.lib	misc_na7_db.lib	--- *)
misc_na7_ir.lib	misc_na7_db_ir.lib	--- *)
misc_na7_fl.lib	misc_na7_db_fl.lib	--- *)
misc_na9.lib	misc_na9_db.lib	--- *)
misc_npc.lib	misc_npc_db.lib	--- *)
tif_na7.lib	tif_na7_db.lib	tif_na7_db_ps.lib
tif_na7_ir.lib	tif_na7_db_ir.lib	tif_na7_db_ps_ir.lib
tif_na7_fl.lib	tif_na7_db_fl.lib	tif_na7_db_ps_fl.lib
tif_na9.lib	tif_na9_db.lib	tif_na9_db_ps.lib
tif_na9_pc.lib	tif_na9_db_pc.lib	tif_na9_db_ps_pc.lib
tif_na9_pc_rv.lib	tif_na9_db_pc_rv.lib	tif_na9_db_ps_pc_rv.lib
tif_npc.lib	tif_npc_db.lib	tif_npc_db_ps.lib
tif_npc_pc.lib	tif_npc_db_pc.lib	tif_npc_db_ps_pc.lib
osx_na7.lib	osx_na7_db.lib	osx_na7_db_ps.lib
osx_na9.lib	osx_na9_db.lib	osx_na9_db_ps.lib

*) The misc library is independent of the partition memory supervision
**) If the Riviera framework is present, the Diagnose and Recovery (DAR) entity API is called in case of fatal errors.

17.2 Configuration Files

xxx is the project name. Templates for configuration files can be found in gpf\template\config.

17.2.1 Xxxcomp.c

17.2.1.1 Task and Entity Configuration

The file xxxcomp.c contains lists of the entities to be created and started.

```
const T_COMPONENT_ADDRESS mmi_list[] =
{
  { mmi_pei_create,       NULL,   ASSIGNED_BY_TI },
```

```
  { NULL,                 NULL,   0 }
};

const T_COMPONENT_ADDRESS cm_list[] =
{
  { sms_pei_create,       NULL,   ASSIGNED_BY_TI },
  { cc_pei_create,        NULL,   ASSIGNED_BY_TI },
  { sm_pei_create,        NULL,   ASSIGNED_BY_TI },
  { ss_pei_create,        NULL,   ASSIGNED_BY_TI },
  { NULL,                 NULL,   (int)"CM" }
};
const T_COMPONENT_ADDRESS rr_list[] =
{
  { rr_pei_create,        NULL,   ASSIGNED_BY_TI },
  { NULL,                 NULL,   0 }
};
```

figure 1

Each of the lists in figure 1 represents one RTOS task and the entities running in this task. The task name will be equal to the entity name. If more than one entity share the same task then the task name has to be set in the last line of the list. In the case entities are grouped to a task, the task priority is equal to the highest prioritized entity, the stack size will be set to the highest entity stack size and the number of queue entries will be set to the highest number of queue entries exported by one of the grouped entities.

The first column contains the address of the pei_create() function of the corresponding entity. The second column is used to create a dummy entity. Dummy entities may be helpful during development. Each of them is running in its own RTOS task, has a message queue and primitives sent to such a dummy entity are discarded including a deallocation of the primitive memory. In the third column the priority that is normally set within the pei_create() function of an entity can be overwritten. This may be useful for object customer who do not have access to the pei_create() function. To modifiy the priority a value between 1 (low) and 255 (high) has to be entered instead of ASSIGNED_BY_TI.

```
const T_COMPONENT_ADDRESS *ComponentTables[]=
{
  mmi_list,
  cm_list,
  rr_list
};
```

figure 2

The task in figure 2 contains all the tasks to be created at startup time.

To create a new entity an entry for this entity it has to be decided whether or not it should reside in its own task or be grouped with different entities in an already existing task. In the first case only an entry for the new entity in one of the entity lists in figure has to be added, in the second case a new list for this task has to be created and the list has to be added to the task list in figure 2.

17.2.1.2 Memory Configuration

In addition to the entity lists and task list xxxcomp.c contains the memory pool and partition pool configuration. Different groups of partition pools are introduced. A partition pool in this case contains a number of fixed size buffers of the same size, a group contains a set of pools.

To create a partition pool the user must provide the number of partitions in this pool as well as the size of the buffers. Also the user needs to provide the memory in which the partition pool is managed. A group of partition pools is declared as an array of partition pool configurations.

```
#define PRIMPOOL_0_PARTITIONS           100
```

```
#define PRIMPOOL_1_PARTITIONS          100
#define PRIMPOOL_2_PARTITIONS           20
#define PRIMPOOL_3_PARTITIONS           20

#define PRIM_PARTITION_0_SIZE           60
#define PRIM_PARTITION_1_SIZE          128
#define PRIM_PARTITION_2_SIZE          632
#define PRIM_PARTITION_3_SIZE         1600
```

The table above shows the configuration for one partition pool group. It contains four partition pools with the partition sizes 60, 128, 632 and 1600 bytes. ATTENTION: The sizes must be in an increasing order.

```
char pool10 [ POOL_SIZE(PRIMPOOL_0_PARTITIONS,ALIGN_SIZE(PRIM_PARTITION_0_SIZE)) ];
char pool11 [ POOL_SIZE(PRIMPOOL_1_PARTITIONS,ALIGN_SIZE(PRIM_PARTITION_1_SIZE)) ];
char pool12 [ POOL_SIZE(PRIMPOOL_2_PARTITIONS,ALIGN_SIZE(PRIM_PARTITION_2_SIZE)) ];
char pool13 [ POOL_SIZE(PRIMPOOL_3_PARTITIONS,ALIGN_SIZE(PRIM_PARTITION_3_SIZE)) ];
```

The required memory size is calculated by the macro POOL_SIZE(). The sizes of the partitions are aligned to values dividable by four via the macro ALIGN_SIZE().

```
const T_FRM_PARTITION_POOL_CONFIG prim_grp_config[] =
{
  { PRIMPOOL_0_PARTITIONS, ALIGN_SIZE(PRIM_PARTITION_0_SIZE), &pool10 },
  { PRIMPOOL_1_PARTITIONS, ALIGN_SIZE(PRIM_PARTITION_1_SIZE), &pool11 },
  { PRIMPOOL_2_PARTITIONS, ALIGN_SIZE(PRIM_PARTITION_2_SIZE), &pool12 },
  { PRIMPOOL_3_PARTITIONS, ALIGN_SIZE(PRIM_PARTITION_3_SIZE), &pool13 },
  { 0                    , 0                    , NULL    }
};
```

The partition pool group configuration is an array of the partition pool configurations.

All partition groups that are defined in this way need to entered in a partition group configuration table.

```
const T_FRM_PARTITION_GROUP_CONFIG partition_grp_config[MAX_POOL_GROUPS+1] =
{
  { "PRIM",       &prim_grp_config[0] },
  { "TEST",       &test_grp_config[0] },
  { "DMEM",       &dmem_grp_config[0] },
  { NULL,         NULL                }
};
```

An array containing the handles of these partition groups needs to be provided

```
T_HANDLE *PoolGroupHandle[MAX_POOL_GROUPS+1] =
{
  &PrimGroupHandle,
  &TestGroupHandle,
  &DmemGroupHandle,
  NULL
};
```

The frame relies on the presence on these the partition pool groups. To create additional pool groups you need to do the following:

- define partition sizes and numbers of the new pool group
- provide the memory to manage the partition pools in
- create an array that represent the new partition group configuration
- add the new partition group the partition group configuration table
- add a handle for the ew group to the pool group handle array

You are allowed to create MAX_POOL_GROUPS partition groups. If this is not sufficient, this definition can be increased without recompilation of the frame. There is no limitation of the number of partition pools in a partition pool group.

The definition of the dynamic memory pool configuration is quite similar to the partition pool configuration. You need to define the sizes of the memory pools.

```
#define EXT_DATA_POOL_SIZE      45000
#define INT_DATA_POOL_SIZE      25000
```

You need to provide the memory which is used for the dynamic memory pools.

```
char ext_data_pool          [ EXT_DATA_POOL_SIZE ];
char int_data_pool          [ INT_DATA_POOL_SIZE ];
```

You need to enter the pool size and the address of the pool memory into a memory pool configuration table.

```
const T_MEMORY_POOL_CONFIG memory_pool_config[MAX_MEMORY_POOLS+1] =
{
  { "INTPOOL", INT_DATA_POOL_SIZE, &int_data_pool[0] },
  { "EXTPOOL", EXT_DATA_POOL_SIZE, &ext_data_pool[0] },
  {  NULL }
};
```

Also you need to provide an array to store the memory pool handles returned at pool creation. The handles can be used to allocate from the memory pools.

```
T_HANDLE *MemoryPoolHandle[MAX_MEMORY_POOLS+1] =
{
  &int_data_pool_handle,
  &ext_data_pool_handle,
  NULL
};
```

You are allowed to create MAX_MEMORY_POOLS memory pool. If this is not sufficient, this definition can be increased without recompilation of the frame.

17.2.2 Xxxinit.c

The file xxxinit.c contains the function that is the entry function of the application. This function is called by the RTOS after it has finished its internal startup. For Nucleus this function is called Application_Initialize(). Here the hardware and/or the drivers can be initialized before the frame is started by calling the function StartFrame().

17.2.3 Xxxdrv.c

The file xxxdrv.c contains a table of drivers used for the test interface.

```
const T_DRV_LIST DrvList[] =
{
  { NULL,       NULL,       NULL,   NULL },
  { "TIF",      TIF_Init,   "TST",  NULL },
  { "TR",       TR_Init,    NULL,   NULL },
  { "SER",      SER_Init,   NULL,   (void*)&SER_DefaultConfig },
  { NULL,       NULL,       NULL,   NULL }
};
```

The first column contains the name of the driver. The second is the address of the inititalization function that is called by the frame during startup. This init function exports the addresses of the functions to access drivers functionality later. The third column contains the name of the process to be notified in the case that something has been received via the test interface. A NULL pointer in this column means that the callback function of the driver above is called. The last column may contain a default configuration string that is passed to the driver after its initialization.

17.2.4 xxxconst.h

The file xxxconst.h contains the constants that determine the dimensions of the frame to be used in the corresponding application. These settings are needed to allocate the memory for the tables to manage these resources. These constants are:

MAX_ENTITIES	Maximum number of entities to be created
MAX_OS_TASKS	Maximum number of RTOS tasks to be created
MAX_SEMAPHORES	Maximum number of semaphores
MAX_COMMUNICATIONS	Maximum number of message queues
MAX_TIMER	Maximum number of timers
MAX_SIMULTANEOUS_TIMER	Maximum number of timers that can run simultaneously
MAX_OSISRS	Maximum number of HSIRs, DFCs
MAX_POOL_GROUPS	Maximum number of partition pool groups
MAX_MEMORY_POOLS	Maximum number of dynamic Memory pools

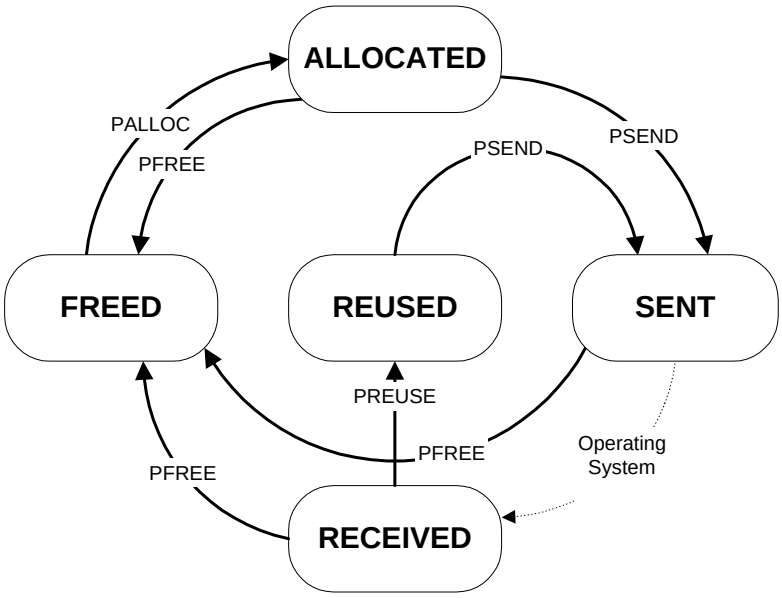
The frame does **not** need to be recompiled when these constants are modified.

18 Partition Pool Monitor

The partition memory pools are used for the exchange of primitives between the entities of the protocol stack. This memory pool group consists of a number of pools with partitions different sizes.

Every time a primitive must be sent, a partition of the needed size is allocated by the sender. The primitive data is stored in this partition and the address of the partition is written into the message queue of the destination process. After evaluation of the primitive, the partition is freed by the receiver. In some cases, the receiver of a primitive may reuse the same partition to send the next primitive. Figure 2 shows the states of a partition and the allowed transitions between these states. The transitions are described in the following.

A partition is set to the state **ALLOCATED** if it is allocated by a body function with the macro PALLOC to hold a primitive that must be sent to another entity. A partition is set to the state **SENT** if it is sent to the destination entity by using the macro PSEND. A partition is in the **RECEIVED** state, if it reached the function pei_primitive() of the destination entity. The state **REUSED** is reached if a received partition is reused by a body function to send a new primitive. It achieves the state **FREED** if a body function calls the macro PFREE after processing the received primitive. The partition state is also freed if a primitive is a redirected to the test interface. In this case, the sending to the original destination is aborted and the partition is freed.


ALLOCATED
PALLOC
PFREE
PSEND
PSEND
FREED
REUSED
SENT
PREUSE
PFREE
PFREE
Operating
System
RECEIVED

18.1 Monitoring

The intention of implementing the Partition Pool Monitor is to supervise the usage of the partition pool as well as to create an instrument to optimize the dimensions of this pool. Both features are enabled by static compiler settings. If MEMORY_SUPERVISION is defined, the error supervision is activated. The OPTIMIZE_POOL option enables the creation of the statistical database for the optimization of the dimensions of the pool. The optimization feature cannot be activated without the pool supervision being enabled

To get an overview over the allocation/deallocation procedures within this pool, all actions are written to a monitoring table. The entries of this table can be read via a test interface and be displayed with PCO.
The monitoring table contains the following information:

- For each of the partitions, the state is stored as described previously. The possible states are FREED, ALLOCATED, SENT and REUSED.
- Each time a partition is allocated, reused, sent or received, the primitive operation code (opc), the source file name and the current line in code is stored.
- For each partition, it is checked if it is mistakenly used by a primitive which does not fit. When a partition is allocated with PALLOC, a partition large enough to hold the primitive to be sent is allocated. This error can only occur when a partition is reused. In the case of an error, the source file name and the current line in code is stored and an error message is traced.
- The transitions between the partition states described previously are monitored. When a transition occurs that is not shown in Figure 2, an error message is traced.
- For each pool within a group of the different partition sizes, there is a class of counters to monitor the allocation activities. Each of these classes consists of one counter for the current number of allocations that is incremented at every allocation and decremented at every deallocation, one for the maximum value of simultaneous allocations, one for the total number of all allocations and two memories to store the current counter when the maximum number of partitions is allocated and when the maximum number of bytes are used from a partition pool. The number of allocated bytes and the number of allocated partitions are supervised with one of these counters.
- Each of the partition sizes is divided into five ranges and for each of these one set of counters is used to monitor the allocation spectrum within these ranges.

18.2 Partition State Messages

- If a partition state transition error occurred (Figure 2.), an error like the following one is sent to the test interface.

  [PPM]: PALLOC – STATE TRANSITION ERROR *(actual state -> new state)* AT *file*, LINE *line*

- If a partition is reused by a primitive that does not fit, an error message of the following type is sent to the test interface.

  [PPM]: PREUSE – OVERSIZE ERROR AT *file*, LINE *line*

- The content of the Partition Monitor Table can be read via the test interface with the system primitive MEMORY. If the mobile is switched off, there is no exchange of primitives between the entities of the protocol stack and no communication with TI controlled Layer 1. For this reason, all partitions of the pool memory must be in the FREED state.

  The output of the partition pool table is done in the following manner:

  command: MEMORY

  response: [PPM]: ALL PARTITIONS FREED

  [PPM]: NO OVERSIZE ERRORS

  This should be the response if everything is working correctly.

If there is a partition which is not freed, the response is as follows:

PPM: PARTITION *prim* STATE *state*, OWNER *owner*, TIME *time*, *file(line)*

If there is a partition in a descriptor list which is not freed, the response is as follows:

PPM: PARTITION in desclist of *prim* STATE *state*, OWNER *owner*, TIME *time, file(line)*

Example

command:

MEMORY

response:

PPM: PARTITION 0x7c60ec, OPC 0x0 STATE ALLOCATED OWNER MMI, TIME 6300, aci_lst.c(108)

PPM: PARTITION 0x7c5dcc, in desclist of OPC 0x220a STATE RECEIVED OWNER LLC, TIME 6350, frame.c(835)

18.3 Optimization of Partition Sizes

The intention of using this tool is to reduce the size of the partition memory pool. The easiest way to do this could be the reduction of the number of partitions in some or each of the pools of one group. An other possible attempt to reach this goal could be the modification of the sizes of the partitions. As a third way, the number of sizes of the pools in the groups could be changed. The best solution probably would be a combination of all of these three possibilities. To get information on how to proceed, an enhanced pool monitoring is available (option OPTIMIZE_POOL).

18.3.1 Features of Enhanced Pool Monitoring

If the enhanced pool monitoring is activated, statistic information about the allocation of partitions is generated. Every time a partition of a specific size is allocated, a byte counter is incremented by the number of bytes needed. Also, a partition counter is incremented. These counters are automatically stored when they reach a maximum and decremented when the specific partition is freed again.

Two values are calculated with the use of these counters. The first is the percentage of allocated partitions to the number of available partitions. The second is the number of bytes requested to the number of bytes really allocated. These two values show the quality of the pool size optimization.

The different partition sizes are divided into five equally sized ranges each. When a maximum of the byte counter or the partition counter is reached the current allocations within these five ranges are stored.

The counter can be read via the test interface as described in the following.

18.3.2 Getting Partition Pool Memory statistic

The output of the partition pool memory statistic is done in the following manner:

command: MEMORY

response:

```
[PPM]: POOL 0 (size 52)
[PPM]: MAXBYTE      bytes      pool 0:  172 bytes      =>  82%
[PPM]: MAXBYTE      partitions pool 0:    0,   0,   1,   2,   1
[PPM]: MAXPART      partitions pool 0:    5 partitions  =>  16%
[PPM]: MAXPART      partitions pool 0:    0,   0,   5,   0,   0
[PPM]: MAXRANGE     partitions pool 0:    0,   0,   5,   2,   2
[PPM]: TOTAL        partitions pool 0:    0,   0,  73,  19,   8
```

This output shows an example for the pool 0 with a partition size of 52 bytes. A similar output is given for each created partition pool.

The ranges for this pool are defined to: range 1 is from 0 bytes to 10 bytes, range 2 from 11...20, range 3 from 21...30, range 4 from 31...40, range 5 from 41...52.

The number in the two lines starting with MAXBYTE mean that in pool 0 with the size of 52 bytes a maximum number of 172 bytes were requested simultaneously. These are 82% of the available bytes in all allocated partitions. In this case, one partition in range 3, two partitions in range 4 and one partition in range 5 was allocated. This information is a measure for the so-called internal fragmentation.

The information in the two lines starting with MAXPART shows that there were a maximum number of five partitions (16% of the available) allocated simultaneously which were all in range 3. This information helps you to dimension the numbers of required partitions of each size.

Within the five ranges, there were a maximum number of simultaneously allocated partitions of 5 in range 3 and 2 in range 4 and 5.

A total number of 73 partitions in range 3, 19 in range 4 and 8 in range 5 were allocated. These numbers help you to adjust the sizes of the required partitions. In case you observe that there were no allocations in range 5, you should consider reducing the partition size from 52 to 40 bytes to save 12 bytes for each created partition.

Attention: In practice there are many use cases and experience showed that it is very difficult or even impossible to adjust the numbers and sizes of the partitions to be optimal for all use cases.

19 Module Specification

The functionality of the frame is split up into different groups of source files.

The frame_xx.lib is build with the following files:

Source File	Functionality
Frame.c	general frame functionality, startup
route.c	primitive routing
prf_func.c	profiler API functions
vsi_pro.c	virtual system interface for task handling
vsi_com.c	virtual system interface for communication
vsi_tim.c	virtual system interface for timer handling
vsi_mem.c	virtual system interface for memory handling
vsi_sem.c	virtual system interface for semaphore handling
vsi_drv.c	virtual system interface for driver handling
vsi_mis.c	virtual system interface for miscellaneous
vsi_ppm.c	virtual system interface for partition supervision
vsi_trc.c	virtual system interface for tracing
os_pro.c	operating system interface for task handling
os_com.c	operating system interface for queue handling
os_tim.c	operating system interface for timer handling
os_mem.c	operating system interface for memory handling
os_sem.c	operating system interface for semaphore handling

os_drv.c	operating system interface for driver handling
os_mis.c	operating system interface for miscellaneous

The misc_xx.lib is build with the following files:

Source File	Functionality
tools.c	type conversions
tok.c	string parsing

The tif_xx.lib is build with the following files:

Source File	Functionality
tst_pei.c	pei interface of TST process
tif.c	layer 3 test interface driver
tr.c	layer 2 test interface driver
emil2.c	EMI test interface driver
ser.c	adaptation to simulated usart or target usart
usart.c	RS232 test interface driver for PC (simulated, NT, 95)
socket.c	socket test interface driver
titrc.c	Riviera test interface driver (calls Riviera trace API functions)

20 Templates

Templates are provided to simplify the implementation of the PEI interface for a new entity and the configuration files needed to set up a new project. These can be found in gpf\template\pei directory.

21 Frequently Asked Questions

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/>