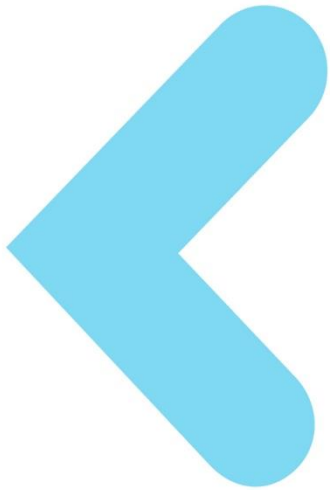
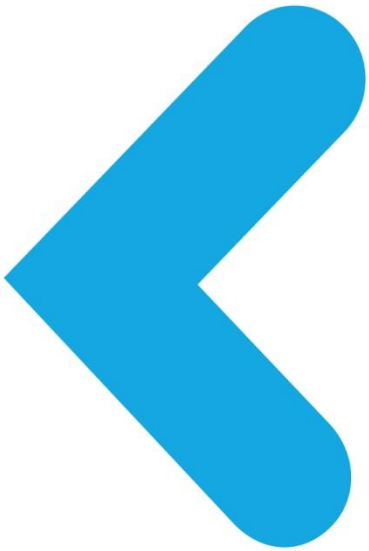
TRUSTONIC



<t-base Driver API Documentation

API Doc.



PREFACE

This specification is the confidential and proprietary information of Trustonic ("Confidential Information"). This specification is protected by copyright and the information described therein may be protected by one or more EC patents, foreign patents, or pending applications. No part of the Specification may be reproduced or divulged in any form by any means without the prior written authorization of Trustonic. Any use of the Specification and the information described is forbidden (including, but not limited to, implementation, whether partial or total, modification, and any form of testing or derivative work) unless written authorization or appropriate license rights are previously granted by Trustonic.

TRUSTONIC MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF SOFTWARE DEVELOPED FROM THIS SPECIFICATION, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. TRUSTONIC SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SPECIFICATION OR ITS DERIVATIVES.

VERSION HISTORY

Version	Date	Modification
1.0	May 6th, 2013	First Issued version for Driver API
1.1	June 20th , 2013	Minor corrections
2.0	November 20th, 2013	Added new functions to support large physical addresses for <t-base-300
2.1	July 22nd, 2014	Deprecated the memory management API with short physical addresses. The 64 bit variants are recommended.
2.2	August 26th, 2014	Added new flags MAP_NOT_SECURE and MAP_STRONGLY_ORDERER which can be used by drApiMapXXX().
2.3	August 27th, 2014	DRAPI_PHYS_MEM_TYPE_HIGH_SECURE is deprecated.
2.4	September 12th, 2014	Documented drApiExtractMsgLen() and drApiExtractMsgCmd() for tlApi_callDriverEx()

TABLE OF CONTENTS

LIST OF TABLES

1 INTRODUCTION

This document specifies the API for developing Secure Drivers running in the <t-base Trusted Execution Environment.

This API is called DrAPI:

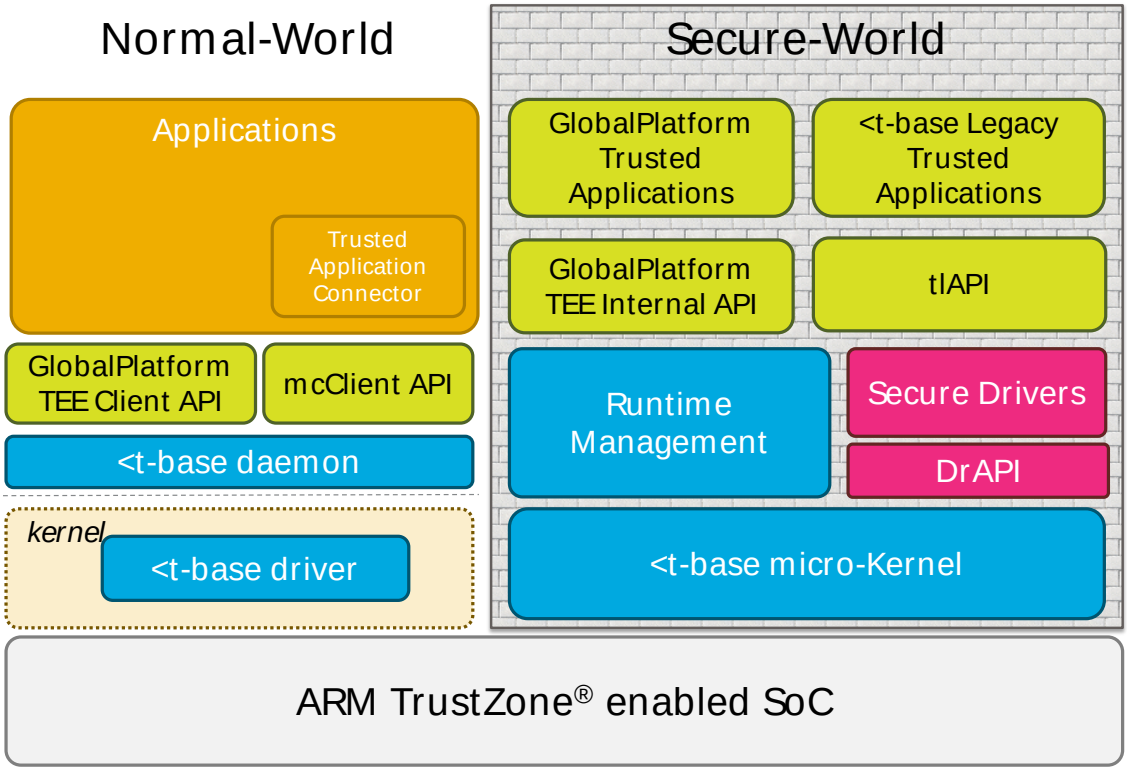


Figure 1: <t-base DrAPI.

For introduction and guidance on how to develop Secure Drivers for <t-base, please refer to the `<t-base Driver Developer's Guide`.

2 API VERSION HISTORY

API Level	Change
Level 1	First <t-base API
Level 2	Added drApiGetClientRootAndSpId Added drApiIpcUnknownMessage Added drApiGetPhysMemType

Level 3	Added drApiMalloc, drApiRealloc, drApiFree
	Added functions for mapping with large physical addresses
	Added drApiCacheDateCleanRange, drApiCacheDateCleanInvalidateRange
	Added drApiRestartThread

3 DRIVER API

3.1 HEADER FILES

The main header file for the Driver API is "DrApi.h".

```
#include "DrApi/DrApi.h"
```

"drStd.h" is required for using standard library types and stack and heap declaration.

```
#include "drStd.h"
```

3.2 COMMON DEFINITION

These definitions are located in "DrApiError.h" and "DrApiCommon.h" files.

3.2.1 Constants

Name	Value	Comment
Driver specific error codes		
DRAPI_OK	0x0	Returns on successful execution of a function
E_DRAPI_KERNEL_ERROR	0xF01	Kernel returned error
E_DRAPI_INVALID_PARAMETER	0xF02	Invalid parameter
E_DRAPI_NOT_PERMITTED	0xF03	Permission error

E_DRAPI_IPC_ERROR	0xF04	Error in IPC
E_DRAPI_TASK_NOT_ACCEPTABLE	0xF05	Task not acceptable for operation
E_DRAPI_CANNOT_MAP	0xF06	Cannot create mapping
E_DRAPI_DRV_NO_SUCH_CLIENT	0xF07	Client does not exist
E_DRAPI_CANNOT_INIT	0xF08	Cannot be initialized
E_DRAPI_NOT_IMPLEMENTED	0xF09	Function not yet implemented
Thread specific error codes corresponding to MTK codes		
E_OK	0	No error
E_INVALID	1	Invalid argument
E_BADTASK	2	Current task does not own target task
E_NOTACTIVATED	3	Task has not been activated
E_NOTOWNER	4	Current task does not own specified task
E_ACTIVATED	5	Task has been activated
E_LIMIT	6	Limit broken
E_NOABILITY	7	No permission
E_STARTED	8	Task or thread have been started
E_BADMAP	9	Invalid mapping (architecture specific error)
E_MAPPED	10	Mapping overlaps existing mapping
E_NOTSTARTED	11	Thread has not been started
E_TIMEOUT	12	Timeout period expired
E_ABORT	13	Operation aborted
E_MSGTYPE	14	Message to send is not of the type the receiver is waiting for
E_MSGLENGTH	15	Message to send exceeds message length the receiver is waiting for
Interrupt mode flags		
INTR_MODE_MASK_TRIGGER	(1U<<0)	Trigger type field
INTR_MODE_TRIGGER_LEVEL	INTR_MODE_MASK_TRIGGER	To trigger on level
INTR_MODE_TRIGGER_EDGE	0	To trigger on edge
INTR_MODE_MASK_CONDITION	(1U<<1)	To trigger condition field

INTR_MODE_CONDITION_FALLING	INTR_MODE_MASK_CONDITION	To trigger on slope condition
INTR_MODE_CONDITION_LOW	INTR_MODE_MASK_CONDITION	To trigger on low level condition
INTR_MODE_CONDITION_RISING	0	To trigger on rise condition
INTR_MODE_CONDITION_HIGH	0	To trigger on high level condition
INTR_MODE_MASK_OCCURANCE	(1U<<2)	Occurrence type field
INTR_MODE_OCCURANCE_ONESHOT	INTR_MODE_MASK_OCCURANCE	To trigger on one shot occurrence
INTR_MODE_OCCURANCE_PERIODIC	0	To trigger on periodic occurrence

Table 1: Driver API Common Constants

Name	Comment
Macros to handle error codes	
DRAPI_ERROR_DETAIL(ecode)	Get detail part of error code
DRAPI_ERROR_MAJOR(ecode)	Get MAJOR part of error code
DRAPI_ERROR_MAJOR_CODE(ecode)	Get MAJOR_CODE part of error code
DRAPI_ERROR_MAJOR_COMPONENT(ecode)	Get MAJOR_COMPONENT part of error code
DRAPI_ERROR_CREATE(ecode, detail)	Create error code: (((ecode)&0xFFF)|((detail&0xFFF)<<12))
Auxiliary macros to handle interrupts	
INTR_MODE_RAISING_EDGE	To trigger on rising edge
INTR_MODE_FALLING_EDGE	To trigger on falling edge
INTR_MODE_LOW_LEVEL	To trigger on low level
INTR_MODE_HIGH_LEVEL	To trigger on high level
Macros used for value to pointer and opposite conversions (used to specify function to execute for Thread API (function drApiStartThread))	
PTR2VAL(p)	Used to obtain value from pointer
VAL2PTR(v)	Used to obtain pointer from value
FUNC_PTR(func)	Used to obtain current function pointer

Table 2: Driver API Common Macros

3.2.2 Types

3.2.2.1 stackEntry_t, stackEntry_ptr, stackTop_pt

```
typedef uint32_t        stackEntry_t;
```

Stack entry type used to statically declare stack in Driver.

```
typedef stackEntry_t    *stackEntry_ptr;
```

Pointer to stack entry type.

```
typedef stackEntry_ptr  stackTop_pt;
```

Pointer to stack entry type. It is used in Thread API (function drApiStartThread).

3.2.2.2 page4KB_t, page4KB_ptr

```
#define SHIFT_4KB               (12U)
#define SIZE_4KB                (1 << SHIFT_4KB)
typedef uint8_t                 page4KB_t[SIZE_4KB];
typedef page4KB_t               *page4KB_ptr;
```

4 KiB page and pointer to 4 KiB page types. The types used by Memory Management API.

3.2.2.3 u32_t, u16_t, u08_t, word_t

```
typedef unsigned int u32_t;
typedef unsigned short u16_t;
typedef unsigned char u08_t;
typedef u32_t word_t;
```

Integer types.

3.2.2.4 drApiResult_t

```
typedef word_t drApiResult_t;
```

Result type used in Driver API functions.

3.2.2.5 taskid_t, *taskid_ptr

```
typedef word_t  taskid_t, *taskid_ptr;
```

Task ID data type and corresponding pointer.

3.2.2.6 threadno_t, *threadno_ptr

```
typedef word_t  threadno_t, *threadno_ptr;
```

Thread number data type and corresponding pointer.

3.2.2.7 threadid_t, *threadid_ptr

```
typedef word_t  threadid_t, *threadid_ptr
```

Thread ID data type and corresponding pointer.

3.2.2.8 intrNo_t, *intrNo_ptr

```
typedef word_t  intrNo_t, *intrNo_ptr
```

Interrupt number type and corresponding pointer.

3.2.2.9 intrMode_t, *intrMode_ptr

```
typedef word_t  intrMode_t, *intrMode_ptr
```

Interrupt mode type and corresponding pointer.

3.3 SYSTEM API

<t-base System API interface provides system information and system functions to Secure Drivers. The Driver System API is declared in **DrApiMcSystem.h** file.

3.3.1 Functions

3.3.1.1 drApiGetVersion

```
_DRAPI_EXTERN_C drApiResult_t drApiGetVersion(
    uint32_t *drApiVersion)
```

Get information about the implementation of the <t-base Driver API version.

Parameters:

- `drApiVersion`: pointer to Driver Api version.

Returns:

- `DRAPI_OK` if version has been set
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code)

3.4 MEMORY MANAGEMENT API

<t-base Memory Management API interface provides memory management functionality to Secure Drivers. The Memory Management API is declared in **DrApiMm.h** file.

3.4.1 Constants

Name	Value	Comment
Memory mapping attributes		
MAP_READABLE	(1U << 0)	Mapping gives the ability to do read access
MAP_WRITABLE	(1U << 1)	Mapping gives have the ability to do write access
MAP_EXECUTABLE	(1U << 2)	Mapping gives have the ability to do program execution
MAP_UNCACHED	(1U << 3)	Mapping gives have uncached memory access
MAP_IO	(1U << 4)	Mapping gives have memory mapped I/O access. Will ignore MAP_UNCACHED, as this would be implied anyway.
MAP_NOT_SECURE	(1U << 7)	Mapping gives the ability to set Non-Secure attribute (Mutual exclusive with MAP_EXECUTABLE). Warning: the memory which is mapped using this flag is Non-Secure and as such must not be trusted, this flag shall be used carefully.
MAP_STRONGLY_ORDERED	(1U << 8)	Mapping gives the ability to access memory with the Strongly Ordered attribute.
Memory type attributes		
DRAPI_PHYS_MEM_TYPE_HIGH_SECURE	(1U<<0)	Deprecated. DRAPI_PHYS_MEM_TYPE_SECURE is returned instead
DRAPI_PHYS_MEM_TYPE_SECURE	(1U<<1)	Secure memory in Dram
DRAPI_PHYS_MEM_TYPE_NON_SECURE	(1U<<2)	NonSecure memory in Dram. Accessible from NonSecure world

Table 3: Driver Memory Management API Constants

3.4.2 Types

3.4.2.1 drApiMarshalingParam_t, *drApiMarshalingParam_ptr

```
#define MAX_MAR_LIST_LENGTH 8
```

```
typedef struct {
    uint32_t functionId;
    union {
        uint32_t parameter[MAX_MAR_LIST_LENGTH];
    } payload;
} drApiMarshalingParam_t, *drApiMarshalingParam_ptr;
```

Marshaled union.

3.4.3 Functions

3.4.3.1 drApiAddrTranslateAndCheck

```
addr_t drApiAddrTranslateAndCheck(addr_t addr)
```

The function performs address translation from Trustlet to Driver address space. It translates an address/pointer given by a Trustlet to the Driver mapping. It also checks for correct address range and null pointer.

Parameters:

- `addr`: Address in Trustlet address space.

Returns:

- In successful case the function returns address in Driver virtual space.
- NULL if address is equal to NULL or if address is out of D3-D8 address space.

3.4.3.2 drApiMapClientAndParams

```
drApiMarshalingParam_ptr drApiMapClientAndParams(
    threadid_t  ipcReqClient,
    uint32_t    params
)
```

The function maps parameters from Trustlet memory space to Driver memory space.

Parameters:

- `ipcReqClient`: Client requesting a service.
- `Params`: Pointer to marshaled parameter in client address space.

Returns:

- Pointer to parameter in the current address space
- NULL in case of any error.

3.4.3.3 drApiMapPhys / drApiMapPhys64

```
drApiResult_t drApiMapPhys(
    const addr_t      startVirt,
    const uint32_t    len,
    const addr_t      startPhys,
```

```
    const uint32_t    attr
)
drApiResult_t drApiMapPhys64(
    const addr_t      startVirt,
    const uint32_t    len,
    const uint64_t    startPhys,
    const uint32_t    attr
)
```

The function maps a physical page to a virtual address. All addresses and lengths must be multiples of page size (4K). The functions allows to access device registers, peripheral memory or any other memory region.

Notice that `drApiMapPhys` is deprecated and kept for backward compatibility. It is recommended to use `drApiMapPhys64` for forward compatibility with systems using large physical addresses.

Parameters:

- `startVirt`: Virtual address in Drivers address space.
- `len`: Length of area.
- `startPhys`: Physical address of hardware.
- `attr`: Mapping attributes (possible values are specified in Table 3: Driver Memory Management API Constants).

Returns:

- `DRAPI_OK` in case of success.
- `E_DRAPI_INVALID_PARAMETER` in case any input parameter is not page size aligned or designated virtual memory area does not fit into D1-D2 address range.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.4.3.4 drApiUnmap

```
drApiResult_t drApiUnmap(
    const addr_t    startVirt,
    const uint32_t  len
)
```

The function removes mapping for a virtual pages. All addresses and lengths must be multiples of page size (4K).

Parameters:

- `startVirt`: Virtual address in task's address space
- `len`: Length of area

Returns:

- `DRAPI_OK` in case of success.
- `E_DRAPI_INVALID_PARAMETER` in case any input parameter is not page size aligned or designated virtual memory area does not fit into D1-D2 address range.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.4.3.5 drApiMapPhysPage4KB / drApiMapPhysPage4KB64

```
drApiResult_t drApiMapPhysPage4KB(
    const page4KB_ptr   virtPage,
    const page4KB_ptr   physPage,
    const uint32_t      attr
)
drApiResult_t drApiMapPhysPage4KB64(
    const page4KB_ptr   virtPage,
    const uint64_t      physPage,
    const uint32_t      attr
)
```

The function maps a single physical page to a virtual address.

Notice that `drApiMapPhysPage4KB` is deprecated and kept for backward compatibility. It is recommended to use `drApiMapPhysPage4KB64` for forward compatibility with systems using large physical addresses.

Parameters:

- `virtPage`: Virtual address in Drivers address space
- `startPhys`: Physical address of hardware
- `attr`: Mapping attributes (are specified in Table 3: Driver Memory Management API Constants).

Returns:

- `DRAPI_OK` in case of success.
- `E_DRAPI_INVALID_PARAMETER` in case any input parameter is not page size aligned or designated virtual memory area does not fit into D1-D2 address range.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.4.3.6 drApiMapPhysPage4KBWithHardware / drApiMapPhysPage4KBWithHardware64

```
drApiResult_t drApiMapPhysPage4KBWithHardware(
    const page4KB_ptr   virtPage,
    const page4KB_ptr   physPage
)
drApiResult_t drApiMapPhysPage4KBWithHardware64(
    const page4KB_ptr   virtPage,
    const uint64_t      physPage
)
```

The function maps a physical page with hardware interface. Actually this is prepared auxiliary function that at first removes mapping of the `virtPage` (if present) and then maps it with MAP_READABLE | MAP_WRITABLE | MAP_IO attributes.

Notice that `drApiMapPhysPage4KBWithHardware` is deprecated and kept for backward compatibility. It is recommended to use `drApiMapPhysPage4KBWithHardware64` for forward compatibility with systems using large physical addresses.

Parameters:

- `virtPage`: Virtual address in Driver address space
- `startPhys`: Physical address of hardware

Returns:

- `DRAPI_OK` in case of success.
- `E_DRAPI_INVALID_PARAMETER` in case any input parameter is not page size aligned or designated virtual memory area does not fit into D1-D2 address range.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.4.3.7 drApiUnmapPage4KB

```
drApiResult_t drApiUnmapPage4KB(
    const page4KB_ptr   virtPage
)
```

The function removes mapping for a single page.

Parameters:

- `startVirt`: Virtual address in Driver address space

Returns:

- `DRAPI_OK` in case of success.
- `E_DRAPI_INVALID_PARAMETER` in case any input parameter is not page size aligned or designated virtual memory area does not fit into D1-D2 address range.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.4.3.8 drApiVirt2Phys / drApiVirt2Phys64

```
drApiResult_t drApiVirt2Phys(
        const taskid_t      taskid,
        const addr_t        virtAddr,
        addr_t *            physAddr
)
drApiResult_t drApiVirt2Phys64(
        const taskid_t      taskid,
        const addr_t        virtAddr,
        uint64_t *          physAddr
)
```

The function converts virtual address (in Driver address space) to physical address.

Notice that `drApiVirt2Phys` is deprecated and kept for backward compatibility. It is recommended to use `drApiVirt2Phys64` for forward compatibility with systems using large physical addresses.

Parameters:

- `taskid`: Reserved for Future Use. It must be set to zero.
- `virtAddr`: Virtual address in Driver address space
- `physAddr`: Physical address

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).
-

3.4.3.9 drApiCacheDataCleanAll

```
drApiResult_t drApiCacheDataCleanAll( void )
```

The function cleans all data cache(s).

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.4.3.10 drApiCacheDataCleanInvalidateAll

```
drApiResult_t drApiCacheDataCleanInvalidateAll( void )
```

The function cleans and invalidates all data cache(s).

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.4.3.11 drApiCacheDataCleanRange

```
drApiResult_t drApiCacheDataCleanRange(
    addr_t *virtAddrStart,
    uint32_t len,
    uint32_t flags
);
```

The function cleans an area of data cache(s).

Parameters:

- `virtAddrStart`: Virtual start address of the range to clean
- `len`: The number of bytes to clean
- `flags`: Which cache levels to clean:
  - DRAPI_CACHE_L1_ONLY
  - DRAPI_CACHE_L1_L2
  - DRAPI_CACHE_ALL

Returns:

- `DRAPI_OK` in case of success.

- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.4.3.12 drApiCacheDataCleanInvalidateRange

```
drApiResult_t drApiCacheDataCleanInvalidateRange(
    addr_t *virtAddrStart,
    uint32_t len,
    uint32_t flags
);
```

The function cleans and invalidates an area of the data cache(s).

Parameters:

- `virtAddrStart`: Virtual start address of the range to clean
- `len`: The number of bytes to clean
- `flags`: Which cache levels to clean:
  - DRAPI_CACHE_L1_ONLY
  - DRAPI_CACHE_L1_L2
  - DRAPI_CACHE_ALL

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.4.3.13 drApiGetPhysMemType / drApiGetPhysMemType64

```
drApiResult_t drApiGetPhysMemType(
        uint32_t *type,
        addr_t addr,
        uint32_t size
        )
drApiResult_t drApiGetPhysMemType64(
        uint32_t *type,
        uint64_t addr,
        uint32_t size
        )
```

The function returns physical memory type (secure or non-secure).

Notice that `drApiGetPhysMemType` is deprecated and kept for backward compatibility. It is recommended to use `drApiGetPhysMemType64` for forward compatibility with systems using large physical addresses.

Parameters:

- `type`: Pointer to address where type is returned. Either DRAPI_PHYS_MEM_TYPE_SECURE or DRAPI_PHYS_MEM_TYPE_NON_SECURE is returned.
- `addr`: start address of checked memory
- `size`: Size checked memory

Returns:

- `DRAPI_OK`

3.4.3.14 drApiMalloc

```
void* drApiMalloc(uint32_t size, uint32_t hint);
```

Allocates a block of memory from the heap.

The address of the allocated block is aligned on a 8-bytes boundary. A block allocated by `drApiMalloc` must be freed by `drApiFree`.

If the size of the space requested is zero, the value returned is still a non-NULL pointer that the Trusted Application must not attempt to access.

Parameters:

- `size`: [in] the number of bytes to be allocated.
- `hint`: [in] must be 0

Returns:

- Upon successful completion, with size not equal to zero, the function returns a pointer to the allocated space. Otherwise, a NULL pointer is returned.

3.4.3.15 drApiRealloc

```
void* drApiRealloc(void* buffer, uint32_t newSize);
```

Reallocates a block of memory from a heap.

This function allows resizing a memory block.

If buffer is NULL, drApiRealloc is equivalent to drApiMalloc.

If buffer is not NULL and newSize is 0, then drApiRealloc is equivalent to drApiFree and returns a non-NULL pointer that the Trusted Application must not attempt to access.

If newSize is less or equal to the current size of the block, the block is truncated, the content of the block is left unchanged and the function returns buffer.

If newSize is greater than the current size of the block, the size of the block is increased. The whole content of the block is copied at the beginning of the new block. If possible, the block is enlarged in place and the function returns buffer. If this is not possible, a new block is allocated with the new size, the content of the current block is copied, the current block is freed and the function returns the pointer on the new block.

Parameters:

- buffer: [in] Pointer to the block of memory that the function reallocates. This value may be null or returned by an earlier call to drApiMalloc or drApiRealloc.
- newSize: [in] size of the memory block in bytes. This value may be zero.

Returns:

- A pointer to the reallocated memory block, a non-NULL pointer if the newSize is zero or NULL if an error is detected.

3.4.3.16 drApiFree

```
void drApiFree(void* buffer);
```

Frees a memory block allocated from a heap by drApiMalloc or drApiRealloc.

This function does nothing if buffer is NULL.

Parameters:

- `buffer`: [in] Pointer to the block of memory to be freed.

3.5 THREAD API

<t-base Driver Thread API interface provides thread handling functionality to Secure Drivers. The Thread API is declared in **DrApiThread.h** file.

3.5.1 Constants

Name	Value	Comment
Common Thread API definitions		
NILTASK	0	It is used for taskid_t type and designates current task
NILTHREAD	0	It is used for threadno_t type and designates current thread
MAX_PRIORITY	(15U)	Maximum priority of a task or thread
TIME_INFINITE	((time_t)((1<< 24) - 1))	Makes infinite time for a task
Control ids for drApiThreadExRegs() API call		
THREAD_EX_REGS_IP	(1U << 0)	Currently set instruction pointer of the thread is replaced by the specified instruction pointer.
THREAD_EX_REGS_SP	(1U << 1)	Currently set stack pointer of the thread is replaced by the specified stack pointer.

Table 4: Driver Thread API Constants

3.5.2 Types

3.5.2.1 time_t, *time_ptr

```
typedef word_t  time_t, *time_ptr ;
```

Time data type.

3.5.3 Functions

3.5.3.1 drApiGetTaskid

```
taskid_t  drApiGetTaskid( void )
```

The function returns task ID for current task.

Returns:

- Task ID for current task.
- 0 in case of any error.

3.5.3.2 drApiTaskidGetThreadid

```
threadid_t drApiTaskidGetThreadid(
    taskid_t taskid,
```

```
    threadno_t threadNo
)
```

The function returns thread ID corresponding to task ID and thread number specified.

Parameters:

- `taskid`: ID of task that owns the thread.
- `threadNo`: Thread number in task.

Returns:

- Thread ID in case of success.
- 0 if task ID or thread number are invalid.

3.5.3.3 drApiGetLocalThreadid

```
threadid_t drApiGetLocalThreadid(
    threadno_t threadNo
)
```

The function returns thread ID for current task corresponding to thread number specified.

Parameters:

- `threadNo`: Thread number in current task

Returns:

- Thread ID in case of success.
- 0 if thread number is invalid.

3.5.3.4 drApiThreadSleep

```
drApiResult_t drApiThreadSleep(
      time_t timeout
)
```

The function makes the calling thread sleep until timeout have elapsed. **At present `timeout` values equal to zero or TIME_INFINITE are only accepted.**

Parameters:

- `timeout`: Time to suspend thread

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.5.3.5 drApiStartThread

```
drApiResult_t drApiStartThread(
    const threadno_t    threadNo,
    const addr_t        threadEntry,
    const stackTop_ptr  stackPointer,
```

```
    const uint32_t      priority,
    const threadno_t    localExceptionHandlerThreadNo
)
```

The function starts a thread in Driver.

Parameters:

- `threadNo`: Thread number in task.
- `threadEntry`: Thread entry function.
- `stackPointer`: Thread top stack pointer (declared statically using DECLARE_STACK).
- `priority`: Thread priority (Maximum level is defined as MAX_PRIORITY, higher priority level corresponds to higher priority thread).
- `localExceptionHandler`: The parameter specifies the number of a thread that serves as an exception handler. (If NILTHREAD is used, an exception will be dispatched to exception handler of task – for Secure Drivers this is RTM exception handler)

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.5.3.6 drApiStopThread

```
drApiResult_t drApiStopThread(
     const threadno_t    threadNo
)
```

The function stops a thread in Driver. If `threadNo` is set to NILTHREAD, the current thread is stopped. The thread that is stopped is detached from any previously attached interrupts. If any thread is waiting for stopped thread to do any IPC, this IPC is aborted.

Parameters:

- `threadNo`: Thread number in task.

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.5.3.7 drApiResumeThread

```
drApiResult_t drApiResumeThread (
    const threadno_t    threadNo
)
```

The function resumes a thread in Driver.

Parameters:

- `threadNo`: Thread number in task.

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.5.3.8 drApiSetThreadPriority

```
drApiResult_t drApiSetThreadPriority(
    const threadno_t    threadNo,
    const uint32_t      priority
)
```

The function sets priority level for a thread in Driver.

Parameters:

- `threadNo`: Thread number in task.
- `priority`: Thread priority (Maximum level is defined as MAX_PRIORITY, higher priority level corresponds to higher priority thread).

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.5.3.9 drApiThreadExRegs

```
drApiResult_t drApiThreadExRegs(
    const threadno_t    threadNo,
    const uint32_t      ctrl,
    const addr_t        ip,
    const addr_t        sp
)
```

The function sets the value registers for specified thread. If THREAD_EX_REGS_IP bit of argument ctrl is set, the currently set instruction pointer is exchanged by the value of the argument ip (3). If THREAD_EX_REGS_SP bit of argument ctrl is set, the currently set stack pointer is exchanged by the value of the argument sp (4).

Parameters:

- `threadNo`: Number of the thread.

- `ctrl`: Control flags
- `ip`: ip value
- `sp`: sp value

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.5.3.10 drApiRestartThread

```
drApiResult_t drApiRestartThread(
    const threadno_t    threadno,
    const addr_t        ip,
    const addr_t        sp
)
```

The function restarts a given thread with given IP and SP.

Parameters:

- `threadNo`: Number of the thread.
- `ip`: ip value
- `sp`: sp value

Returns:

- `DRAPI_OK` in case of success.

3.6 INTERRUPT API

<t-base Driver Interrupt API interface provides interrupt managements functionality to Secure Drivers. The Interrupt API is declared in **DrApiThread.h** file.

3.6.1 Functions

3.6.1.1 drApiIntrAttach

```
drApiResult_t drApiIntrAttach(
    intrNo_t intrNo,
    intrMode_t intrMode
)
```

The function attaches an interrupt with the specified trigger condition to current thread. Please refer to the target platform specific <t-base documentation about the trigger modes supported for each interrupt. In most cases, the mode parameters will be INTR_MODE_RAISING_EDGE, as interrupts usually indicate that a certain event has happened.

Parameters:

- `intrNo`: Interrupt number
- `intrMode`: Interrupt mode (possible values are specified in Table 1: Driver API Common Constants).

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.6.1.2 drApiIntrDetach

```
drApiResult_t drApiIntrDetach(
      intrNo_t intrNo
)
```

The function detaches interrupt from current thread.

Parameters:

- `intrNo`: Interrupt number

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.6.1.3 drApiWaitForIntr

```
drApiResult_t drApiWaitForIntr(
    const intrNo_t  intrNo,
```

```
    const uint32_t  timeout,
    intrNo_t        *pIntrRet
)
```

The function waits with specified timeout for interrupt message from kernel.

Parameters:

- `intrNo`: Interrupt number (if ANYINTR is used, the interrupt is returned in the parameter intrRet).
- `timeout`: Timeout to wait (allowed the same values as for MTK signal_wait()).
- `pIntrRet`: The number of interrupt occurred (parameter can be NULL if caller does not need this).

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.6.1.4 drApiTriggerIntr

```
drApiResult_t drApiTriggerIntr(
    intrNo_t intrNo
)
```

The function triggers software interrupt in the NWd to notify it.

Parameters:

- `intrNo`: Interrupt number.

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.7 IPC API

<t-base Driver IPC API interface provides inter process communication facilities to Secure Drivers. The IPC API is declared in **DrApiIpcMsg.h** file.

3.7.1 Types

3.7.1.1 message_t

Possible message types/event types of the system.

Enumerator:

- MSG_NULL: Used for initializing state machines
- MSG_RQ: Request (client -> server) (tlApi_callDriver())
- MSG_RQ_EX: Request (client -> server) (tlApi_callDriverEx())
- MSG_RS: Response (server -> client)
- MSG_RD: Ready (server -> IPCH)
- MSG_NOT: Notification (client -> IPCH)
- MSG_CLOSE_TRUSTLET: Close Trustlet (MSH -> IPCH; IPCH -> all servers)
- MSG_CLOSE_TRUSTLET_ACK: Close Trustlet Ack (servers -> IPCH)
- MSG_MAP: Map (Driver <-> IPCH)
- MSG_ERR_NOT: Error Notification (EXCH/SIQH -> IPCH)
- MSG_CLOSE_DRIVER: Close Driver (MSH -> IPCH; IPCH -> Driver)
- MSG_CLOSE_DRIVER_ACK: Close Driver Ack (Driver -> IPCH; IPCH -> MSH)
- MSG_GET_DRIVER_VERSION: GetDriverVersion (client <-> IPCH)
- MSG_GET_DRAPI_VERSION: GetDrApiVersion (Driver <-> IPCH)
- MSG_SET_NOTIFICATION_HANDLER: Set (change) the SIQ handler thread (Driver <-> IPCH)
- MSG_GET_REGISTRY_ENTRY: Get registry entry (Driver <-> IPCH)
- MSG_DRV_NOT: Notification (Driver -> Trustlet)
- MSG_SET_FASTCALL_HANDLER: Fastcall handler installation <-> Trustlet
- MSG_GET_CLIENT_ROOT_AND_SP_ID: Driver <-> IPCH

3.7.2 Functions

3.7.2.1 drApiIpcWaitForMessage

```
drApiResult_t drApiIpcWaitForMessage(
    threadid_t        *pIpcPartner,
    uint32_t          *pMr0,
    uint32_t          *pMr1,
```

```
    uint32_t           *pMr2
)
```

The function waits with infinite timeout for IPC message.

Parameters:

- `ipcPartner`: IPC partner to signal.
- `pMr0`: IPC register 0.
- `pMr1`: IPC register 1.
- `pMr2`: IPC register 2.

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.7.2.2 drApiIpcCallToIPCH

```
drApiResult_t drApiIpcCallToIPCH(
    threadid_t        *pIpcPeer,
    message_t         *pIpcMsg,
    uint32_t          *pIpcData
)
```

The function sends ready message or answer to IPCH and waits for a client request.

Parameters:

- `ipcPeer`: Destination to send message to.
- `ipcMsg`: IPC message.
- `ipcData` Additional IPC data.

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.7.2.3 drApiIpcSignal

```
drApiResult_t drApiIpcSignal(
    const threadid_t  receiver
)
```

The function sets signal. The signal (SIGNAL) is used by a thread to inform another thread about an event. The signal operation is asynchronous, which means that the operation will return immediately without blocking the user. Function uses auto-

clear signals, meaning that the signal is cleared automatically when the receiver receives it.

Parameters:

- `receiver`: Thread to set the signal for.

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.7.2.4 drApiIpcSigWait

```
drApiResult_t drApiIpcSigWait( void )
```

The function runs signal wait operation. A thread uses the operation to check if a signal has occurred. If no signal is pending the thread will block until a signal arrives.

Returns:

- DRAPI_OK in case of success.
- Any combination of DRAPI_ERROR_CREATE(Driver specific error code, MTK error code).

3.7.2.5 drApiNotify

```
drApiResult_t drApiNotify ( void )
```

The function notifies NWd driver.

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.7.2.6 drApiSyscallControl

```
drApiResult_t drApiSyscallControl(
    uint32_t controlid,
    uint32_t param1,
    uint32_t param2,
    uint32_t param3,
    uint32_t param4,
    uint32_t *data
)
```

The function makes control syscall with given parameters.

Parameters:

- `controlid`: Control ID.
- `param1`: Parameter 1.
- `param2`: Parameter 2.
- `param3`: Parameter 3.
- `param4`: Parameter 4.
- `data`: Is set by control syscall

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.7.2.7 drApiReadOemData

```
drApiResult_t drApiReadOemData(
        const uint32_t  offset,
        uint32_t        *data
)
```

The function reads OEM data starting from given offset.

Parameters:

- `offset`: Data offset.
- `data`: Is set by control syscall

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.7.2.8 drApiNotifyClient

```
drApiResult_t drApiNotifyClient(
    const threadid_t  client
)
```

The function sends notification to client.

Parameters:

- `client`: Client's thread ID

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.7.2.9 drApiGetClientRootAndSpId

```
drApiResult_t drApiGetClientRootAndSpId(
    uint32_t        *rootId,
    uint32_t        *spId,
    const threadid_t  client
)
```

The function gets ID of Root and Service Provider ID of the specified client.

Parameters:

- `rootId`: ID of Root
- `spId`: Service Provider ID
- `client`: Thread ID

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.7.2.10 drApiIpcUnknownMessage

```
drApiResult_t drApiIpcUnknownMessage(
    threadid_t       *pIpcPeer,
    message_t        *pIpcMsg,
    uint32_t         *pIpcData
)
```

This function handles unknown messages. It has to be called by Driver if it receives a message it does not recognize.

Parameters:

- `pIpcPeer`: Sender of message.
- `pIpcMsg`: IPC message.
- `pIpcData`: Additional IPC data.

Returns:

- `DRAPI_OK` in case of success.
- Any combination of `DRAPI_ERROR_CREATE`(Driver specific error code, MTK error code).

3.7.2.11 drApiExtractMsgLen

```
uint32_t    drApiExtractMsgLen(uint32_t msg)
```

This function extracts the size of the payload sent to the driver when the client is calling `tlApi_callDriverEx`();

Parameters:

- `msg`: message returned by `drApiIpcCallToIPCH`() combining the length of the payload and the message command .

Returns:

- The size of the payload.

3.7.2.12 drApiExtractMsgCmd

```
uint32_t    drApiExtractMsgCmd (uint32_t msg)
```

This function extracts the command sent to the driver when the client is calling `tlApi_callDriverEx`();

Parameters:

- `msg`: message returned by `drApiIpcCallToIPCH`() combining the length of the payload and the message command .

Returns:

- The command (one of the MSG_XXX defined in 3.7.1.1).

3.8 LOGGING API

The <t-base Driver Logging API interface provides logging functions to Secure Drivers. The Logging API is declared in **DrApiLogging.h** file.

3.8.1 Functions

3.8.1.1 drApiLogvPrintf, tlApiLogvPrintf

```
_DRAPI_EXTERN_C void drApiLogvPrintf(
        const char *fmt,
        va_list args);
```

```
_DRAPI_EXTERN_C void drApiLogPrintf(
        const char *fmt,
        ...)
```

Formatted logging functions. Minimal printf-like function to print logging message to NWd log.

Supported formatters:

- %s String, NULL value emit "<NULL>".
- %x %X hex
- %p pointer (hex with fixed width of 8)
- %d i signed decimal
- %u unsigned decimal
- %t timestamp (if available in platform). NOTE: This does not consume any value in parameter list.
- %% outputs single %
- %s, %x, %d, and %u support width (example %5s). Width is interpreted as minimum number of characters. Hex number is left padded using '0' to desired width. Decimal number is left padded using ' ' to desired width. String is right padded to desired length.

Newline is used to terminate logging line.

Parameters:

- `fmt`: Formatter.
- `args`: Argument list.

Macros `drDbgPrintLnf` that just adds EOL symbol to `drDbgPrintf` function is provided in addition.