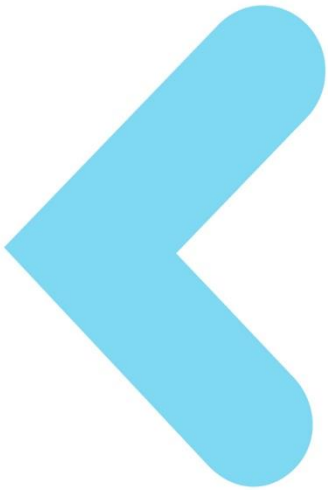
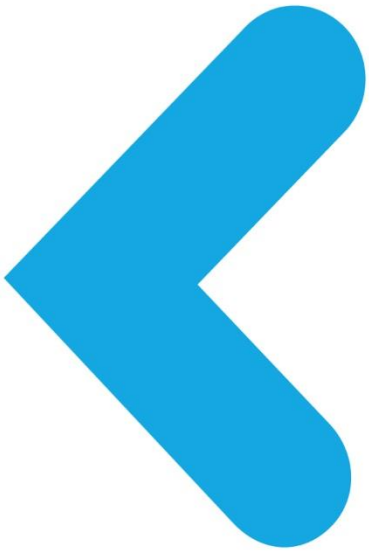
TRUSTONIC



Kinibi Driver Developer’s Guide



PREFACE

VERSION HISTORY

Version	Date	Modification
1.0	26 Feb 2013	First version
2.0	November 19th 2013	Updated for Kinibi-300
2.1	June 9th, 2014	Updated for Kinibi-301
2.2	September 10th 2014	Updated for Kinibi-301B
2.3	September 12th 2014	Documented tlApi_callDriverEx() in section 5.3
2.4	November 18th, 2014	Updated for Kinibi-302A • Introduced extended memory layout • Floating Point support • Increase TEE virtual space to 128MB.
2.5	August 12th, 2015	Updated for Kinibi-310A
2.6	October 29th, 2015	Updated for Kinibi-310B • Introduce stack protection

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This Developer's Guide is a practical introduction for developing Secure Drivers for Kinibi.

This guide provides an overview of the different sets of API available and explains how to use the Trustonic DDK.

The full API for Secure Drivers can be found in the Kinibi Driver API Documentation.

2 Kinibi PRODUCT OVERVIEW

Kinibi is a portable and open Trusted Execution Environment (TEE) aiming at executing Trusted Applications (TA) on a device. It includes also built-in features like cryptography or secure objects. It is a versatile environment that can be integrated on different System on Chip (SoC) supporting the ARM TrustZone technology.

Kinibi uses ARM TrustZone to separate the platform into two distinct areas, the Normal-World with a conventional rich operation system and rich applications and the Secure-World.

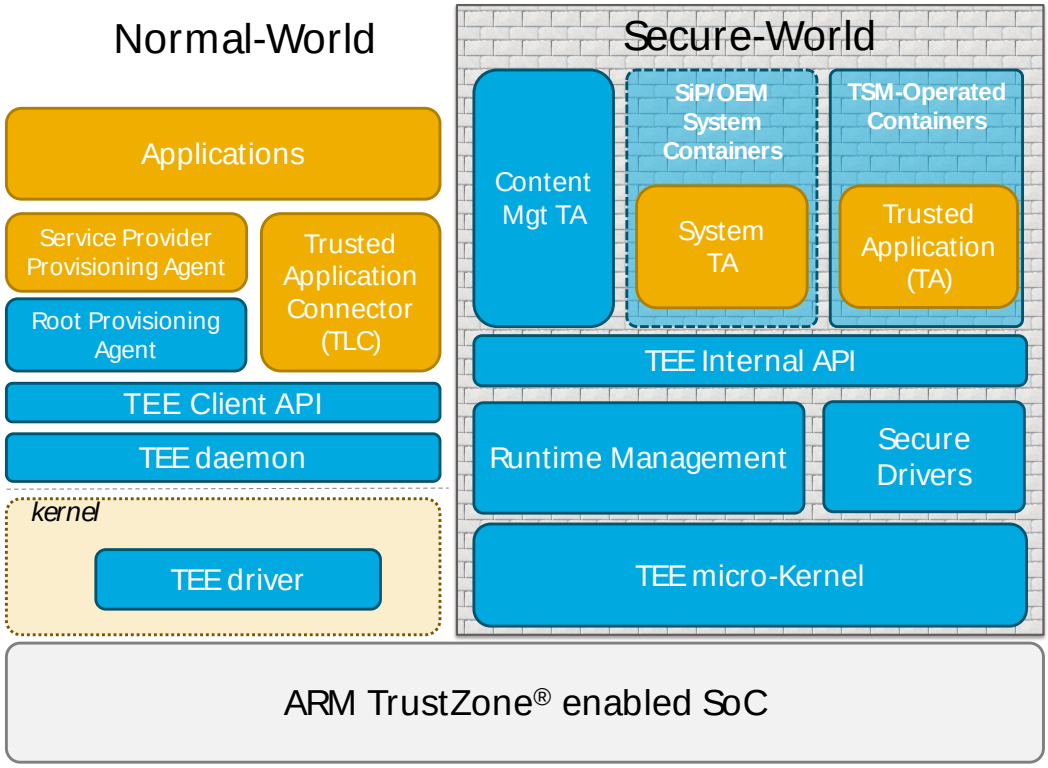


Figure 1: Kinibi Architecture Overview.

The Secure-World contains essentially the Kinibi core operating system and the Trusted Applications. It provides security functionality to the Normal-World with an on-device client-server architecture. The Normal-World contains mainly software which is not security sensitive (the sensitive code should be migrated to the Secure-World) and it calls the Secure-World to get security functionality via a communication mechanism and several APIs provided by Kinibi. The caller in the Normal-World is usually an application, also called a client.

The Trusted Applications in the Secure-World are installed in Containers. A Container is a security domain which can host several Trusted Applications controlled by a third party. There are two kinds of Containers:

- The TSM-Operated Containers, which are created at runtime under the control of Trustonic. Trusted Applications of a TSM-Operated Container can be administrated Over-The-Air via a Trusted Service Manager (TSM).

- The System Container, which is pre-installed at the time of manufacture along with some Trusted Applications. Trusted Applications in the system container cannot be downloaded or updated Over-The-Air via a TSM.

The root Provisioning Agent is a Normal-World component which communicates between the Device and TRUSTONIC's backend system to create TSM-Operated Containers within Kinibi at runtime.

Note that in order to enable Kinibi and the TSM-Operated Containers on a Device, the OEM must install Trustonic's Key Provisioning Host (KPH) at its manufacturing line. The KPH is a tool which injects a key on the device and which stores a copy in the Trustonic backend system.

Kinibi provides APIs in the Normal-World and the Secure-World. In the Normal-World, the Kinibi Client API is the API to communicate from the Normal-World to the Secure-World. This API enables to establish a communication channel with the Secure-World and send commands to the Trusted Applications. It enables also to exchange some memory buffers between the Normal-World and the Trusted Applications.

In the Secure-World, Kinibi provides the Kinibi Internal API which is the interface to be used for the development of Trusted Applications.

Furthermore, the Kinibi architecture supports Secure Drivers to interact with any secure peripherals. Trusted Applications and Secure Drivers use native execution on the ARM platform and can use the Floating Point and NEON instruction sets.

2.1 KINIBI API FOR DRIVER DEVELOPERS

Kinibi is an open environment which provides API for developers to develop Trusted Applications and Client Applications.

Kinibi supports Secure Drivers to address security peripherals in order to enable built in features such as DRM or Trusted User Interface. Secure Drivers can also be used to provide additional custom features to Trusted Applications.

Secure Drivers are developed using the DrAPI.

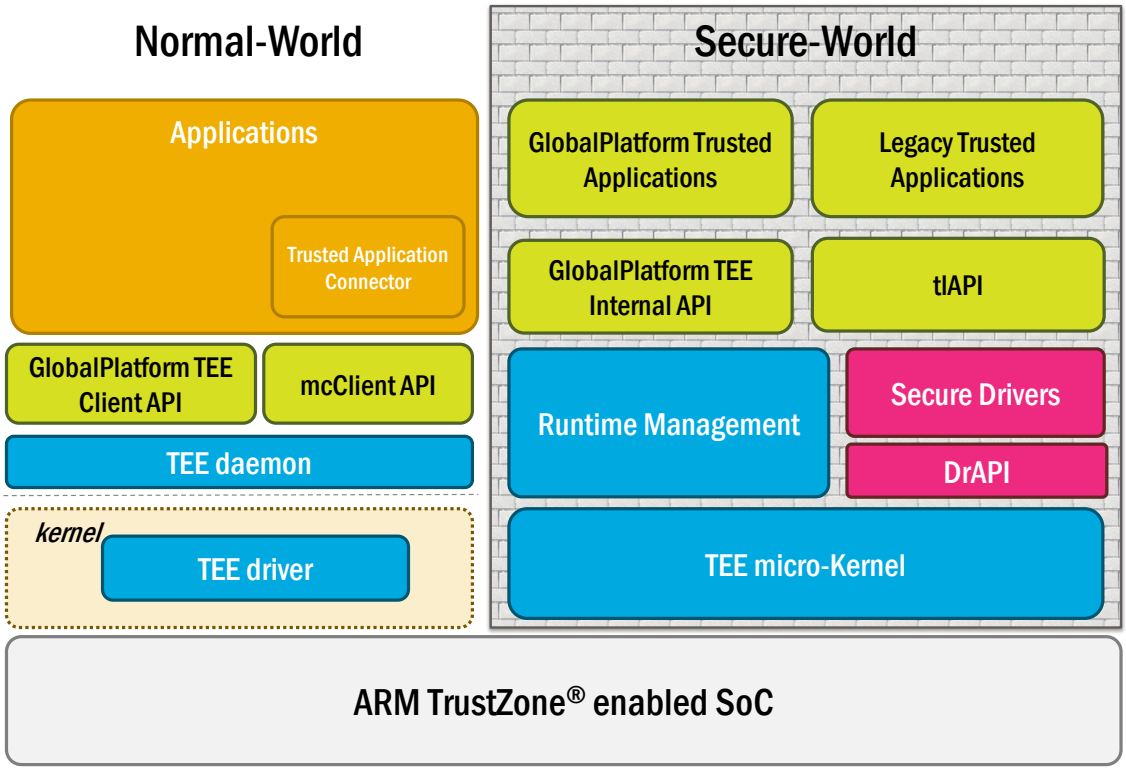


Figure 2: Kinibi API Overview.

3 ABOUT SECURE DRIVERS

A Secure Driver enables a Trusted Application to access peripherals in a controlled and coordinated manner. Secure Drivers hide the hardware complexity from the Trusted Applications.

Kinibi uses a microkernel architecture and as such Secure Drivers are implemented as separate processes. Basically a Secure Driver is a user mode task similar to a Trusted Application. However, a Secure Driver has a different interface to Kinibi which allows accessing the hardware.

3.1 LOADING A SECURE DRIVER

Kinibi supports dynamic loading of Secure Drivers. Secure Drivers are loaded and authenticated into Kinibi in the same way as the System Trusted Applications. Alternatively Secure Drivers can be loaded by running Kinibi Normal World daemon with the '-r' option.

3.2 SECURE DRIVER INTERFACES

Secure Drivers can use more system calls than Trusted Applications and thereby control hardware. The Kinibi system calls are available via the Driver API. Drivers cannot currently talk to each other.

The mechanism for communicating between Secure Drivers and Trusted Applications is based on message passing IPC with marshalling of shared memory.

3.3 LIMITATIONS

Note the following limitations concerning the usage of Secure Drivers:

- Secure Drivers cannot use the API for Trusted Applications.
- Secure Drivers cannot communicate between each other.

Most of the use cases will have an architecture that consists of a Client Application, a Trusted Application and a Secure Driver. The Client Application orchestrates everything and exchanges encrypted objects with the Trusted Application. The Trusted Application will decrypt them and pass commands to the Secure Driver. The Secure Driver will take commands from the Trusted Application and transfer data in and out of the device.

3.4 SECURE DRIVER FEATURES AND SECURITY

3.4.1 Processing Environment

A Secure Driver runs in a special virtual address space that separates it from other Secure Drivers, from Trusted Applications, from Kinibi kernel and runtime, and from device memory.

Secure Drivers can only use the Driver Api.

3.4.2 Memory Management

Secure Drivers use similar address space layout as Trusted Applications, see the Kinibi Developers Guide for more information. In particular, a Secure Driver has to declare its stack size and heap size in the same way as Trusted Applications.

The developer has to use the following macro to declare the required stack size:

```
DECLARE_DRIVER_MAIN_STACK(4096);
```

With API LEVEL < 8, the Secure Driver stack is also part of the program data and is statically allocated during compile time. Kinibi-310B introduces API LEVEL 8 and MMU-protected main stack. When you select level 8, the binary only contains an integer with the minimum stack size. The startup code of the application will allocate a stack with one unmapped MMU page before and after the stack area. That way, stack overflows and underflows will not silently overwrite global variables and heap, but cause a segmentation fault that helps discover stack problems during the development phase.

A Secure Driver can use common heap functionality using drApiMalloc, drApiFree and drApiRealloc to organize its dynamic memory.

- Legacy memory layout

The heap is also allocated during compile time and part of the Secure Driver program data. The developer has to use the following macro to reserve the required heap size.

```
DECLARE_DRIVER_MAIN_HEAP(16384);
```

- Extended memory layout

The heap is allocated during the loading of the Secure Driver. The developer has to set the following lines in the makefile:

```
HEAP_SIZE_INIT := 4096 # for example
HEAP_SIZE_MAX := 8192  # optional
```

To access device registers and peripheral memory, a Secure Driver can map physical memory into its virtual address space using the Driver API.

- A Secure Driver can map physical memory to access device registers
- A Secure Driver can map secure or non-secure memory regions
- A Secure Driver can translate its own virtual address to physical address by using a dedicated API mentioned below.
- A Secure Driver can do L1 d-cache clean/clean invalidate operations.
- To access data residing in Trusted Application memory, a Secure Driver may map the entire Trusted Application memory to its virtual address space. This is required for parameter passing to and from a Trusted Application. There is a function in Driver API that can be used for this purpose.

The Extended Memory Layout

Kinibi-302A introduces an extended memory layout for Trusted Applications and Secure Drivers. With the extended memory layout, the Trusted Applications and Secure Drivers can use more than 1MB of code and data and their optional heap can also be extended.

By default, this extended memory layout is not activated and must be explicitly requested in the makefile.

When using the extended memory layout, Secure Drivers must map memory using the following dedicated functions: drApiMapTaskBuffer(), drApiMapPhysicalBuffer(), drApiUnmapBuffer() and drApiUnmapTaskBuffers().

The compatibility between Trusted Applications and Secure Drivers using different memory layouts is as follow:

- A Trusted Application using the legacy memory layout **can** call a Secure Driver using the legacy memory layout.
- A Trusted Application using the extended memory layout **cannot** call a Secure Driver using the legacy memory layout.
- A Trusted Application using the extended memory layout **can** call a Secure Driver using the extended memory layout.
- A Trusted Application using the legacy memory layout **can** call a Secure Driver using the extended memory layout.

It is highly recommended to use the extended memory layout when writing new Secure Drivers.

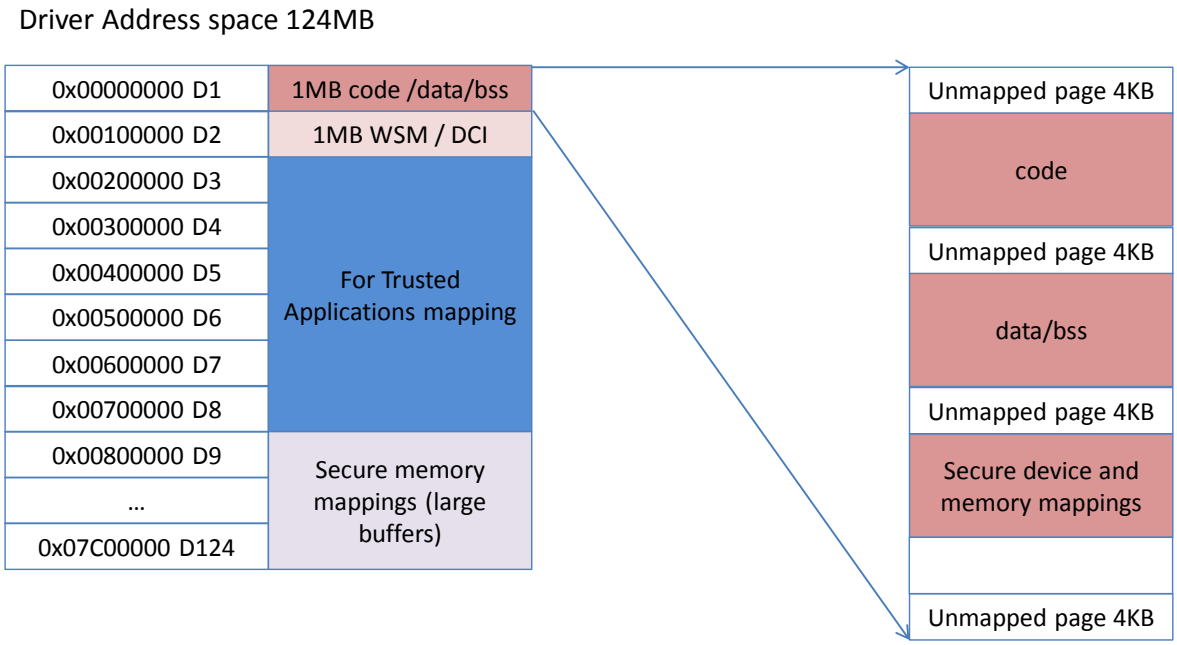


Figure 3: Secure Driver Virtual Address Space with Legacy layout.

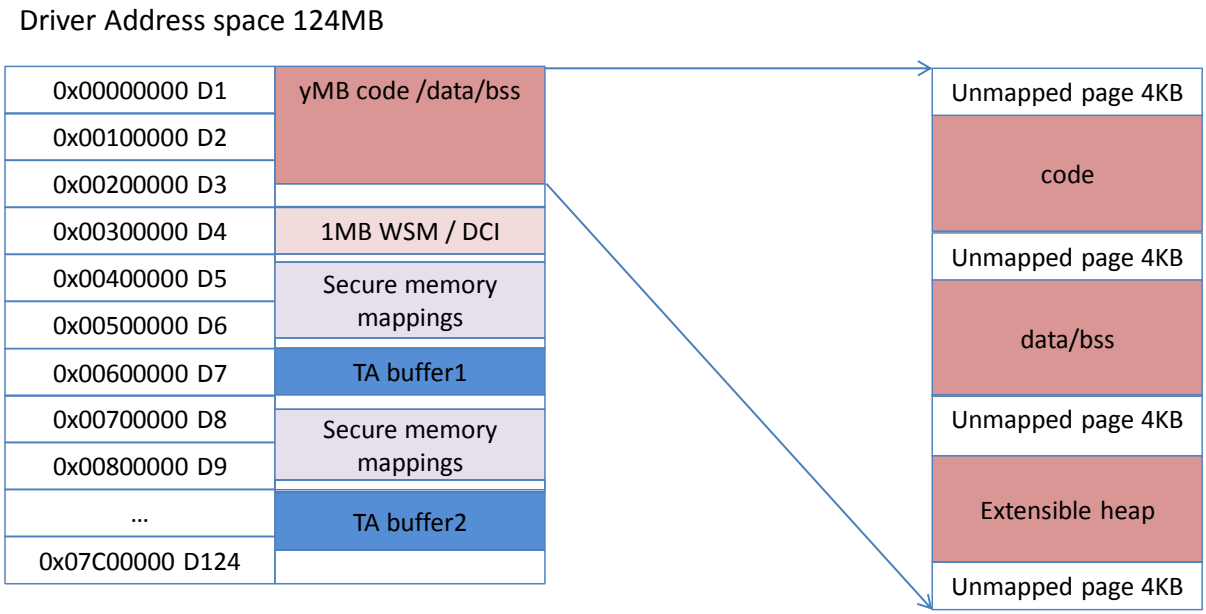


Figure 4: Secure Driver Virtual Address Space with Extended layout.

3.4.3 IPC

The communication between a Trusted Application and a Secure Driver must be done with an IPC. The Driver IPC API can only be used to pass notifications with one register of payload. Usually, this payload is a pointer to a buffer in the Trusted Application address space, with the size of the buffer decided in advance. If a variable-size buffer is needed, it is possible to use a buffer made of a first, constant-size data structure containing the length of a second, variable-size part. Command IDs and data structures can be used to marshal call parameters onto this buffer.

3.4.4 Parameter Verification

A Secure Driver must validate all data that it gets from a Trusted Application. For example pointers to memory areas must be verified.

All given length parameters must be checked to not to exceed memory areas.

When the extended memory layout is used, the mapping API is doing this verification automatically.

3.4.5 Interrupts, Threads, Exceptions and Messages

A Secure Driver starts off with one thread.

To wait for an IPC, a Secure Driver must call a blocking Driver API function.

To wait for interrupts, a Secure Driver must call a blocking Driver API function.

A Secure Driver can create multiple threads to allow waiting for various events.

A Secure Driver can send IPC between threads for synchronization.

In addition to IPC from Trusted Applications, a Secure Driver also has to handle system messages (e.g. Trusted Application close messages received from Kinibi)

A Secure Driver can handle exceptions in its threads. Therefore an exception handler thread must be installed that waits for exceptions and restarts the failing threads.

3.4.6 Security

Secure Drivers are security-critical components and should not allow an attacker to compromise Kinibi security. In particular:

- Secure Drivers should avoid complexity and large size.
- Beware of race conditions (i.e. when a Trusted application or Normal World manipulates data while being processed by secure driver).
  - In particular, do not read the same value twice from shared memory: the value might have changed. Copy, then validate, then use.
- Don't map memory regions belonging to Kinibi, only regions belonging to the hardware that the driver is for.
- Validate all input data received from the Normal World and Trusted Applications.
  - Don't allow TAs to access arbitrary memory regions.
  - Make proper address checks to ensure that secure driver doesn't read/write to arbitrary memory regions.
  - Implement boundary checks (buffers and sizes) to ensure that buffers passed by TAs or Normal World remain within valid memory regions.
  - If writing to a shared memory, make sure that memory is not read-only.
- Use memory mapping attributes (MAP_READABLE, MAP_WRITABLE, MAP_EXECUTABLE, MAP_UNCACHED) properly while mapping physical memory.
  - Mapping with MAP_EXECUTABLE flag will allow code execution from that particular mapped memory region if exploited. It should normally not be used in a driver.
- When deleting a data from a particular mapped memory region, remember that data may be cached, meaning that data may still remain in main memory.

- If memory region is marked as cached memory region, remember to perform proper cache maintenance operations.
- Secure drivers can alternately map memory region as uncached (can have performance implications depending in data size).

- While using peripherals, remember to handle registers and buffers securely to ensure that sensitive data is not leaked. In particular, if a peripheral is shared between the normal world and the secure world, make sure that the secure world locks the normal world out, initializes the peripheral properly.
  - Make sure that after completing the request, registers and buffers of secure peripherals are cleaned properly, before releasing the peripheral to the normal world if applicable.
- While doing arithmetic operations, take care to avoid or detect overflows.

3.4.7 Availability

- Secure Drivers should react to all system messages appropriately and in a timely manner.
- Secure Drivers should avoid polling.

3.4.8 Instances and Sessions

There can be only one instance of each Secure Driver.

A Secure Driver exports a Driver ID and Trusted Applications can call this Driver ID.

A Secure Driver can handle only one request from a Trusted Application at a time. A Secure Driver can implement sessions for Trusted Applications and use Trusted Application ID.

4 IMPLEMENTING A SECURE DRIVER

A Secure Driver consists of:

- A main thread that works as an exception handler thread
- An IPC handler thread for handling IPC messages from Trusted Applications and system messages from <-base
- An ISR handler thread if there is a need to attach to an interrupt and wait for that interrupt to occur

4.1 DRIVER MAIN THREAD/EXCEPTION HANDLER THREAD

Depending on the purpose of the Secure Driver, the main thread should implement at least following functionalities:

- Initialization of the task/thread control
- Starting IPC and ISR handler threads
- Implementing exception handler loop for handling exceptions caused by other local threads and restarting them when possible
- Taking care of needed HW initialization
- Support for power saving mode whenever the task is idle

4.2 IPC HANDLER THREAD

The task of the IPC handler thread is to handle incoming IPC messages and process them accordingly.

When IPC thread starts, it needs to send ready message ('MSG_RD' notification) to Kinibi (only when called first time) and then wait IPC messages by calling the following Driver API:

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

4.2.1 Example with the extended memory layout.

```
for (;;)
{
    /*
     * When called first time sends ready message to IPC server and
     * then waits for IPC requests
     */
    if (E_OK != drApiIpcCallToIPCH(&ipcClient, &ipcMsg, &ipcData))
    {
        continue;
    }

    /* Dispatch request */
```

```
    switch (drApiExtractMsgCmd(ipcMsg))
    {
    case MSG_CLOSE_TRUSTED APPLICATION:
        /**
         * Trusted Application close message
         */
        ipcMsg = MSG_CLOSE_TRUSTED APPLICATION_ACK;
        ipcData = TLAPI_OK;
        break;
    case MSG_CLOSE_DRIVER:
        /**
         * Driver close message
         */
        ipcMsg = MSG_CLOSE_DRIVER_ACK;
        ipcData = TLAPI_OK;
        break;
    case MSG_GET_DRIVER_VERSION:
        /**
         * Driver version message
         */
        ipcMsg = (message_t) TLAPI_OK;
        ipcData = DRIVER_VERSION  ;
        break;
    case MSG_RQ_EX:
        /* init tlRet value */
        tlRet = TLAPI_OK;
        /**
         * Handle incoming IPC requests via TL API.
         */
        drRet = drApiMapTaskBuffer(THREADID_TO_TASKID(ipcClient),
                                   (addr_t )ipcData,
                                   sizeof(drMarshalingParam_t),
                                   MAP_READABLE|
                                   MAP_WRITABLE,
                                   (void **)&pMarshal);
        if (drRet != DRAPI_OK)
        {
            continue;
        }
        /*  The  marshaling  structure  is  mapped  with  the  secure
attribute */
        /* Process the request */
        switch (pMarshal->functionId)
        {
            case FID_DR_SAMLE01:
                 /**
                  * Handle Sample01 request accordingly
                  */
            drRet = drApiMapTaskBuffer(THREADID_TO_TASKID(ipcClient),
                        pMarshal->buffer1,
                        pMarshal->length1,
                        MAP_READABLE|MAP_WRITABLE|MAP_ALLOW_NONSECURE,
                        (void **)&buffer1);
                 if (drRet != DRAPI_OK)
                 {
```

```
                    continue;
                }
                /* The driver can now access to non-secure buffer1 */
                break;
            default:
                /* Unknown message has been received*/
                tlRet = E_TLAPI_UNKNOWN_FUNCTION;
                break;
        }
        /* Update response data */
        ipcMsg  = MSG_RS;
        ipcData = tlRet;
        /* Unmap all buffers for this client task */
        drApiUnmapTaskBuffers(THREADID_TO_TASKID(ipcClient));
        break;
    default:
        /* Unknown message has been received*/
        ipcMsg  = MSG_RS;
        ipcData = E_TLAPI_DRV_UNKNOWN;
        break;
    }
}
```

4.2.2 Example with the legacy memory layout:

```
for (;;)
{
    /*
     * When called first time sends ready message to IPC server and
     * then waits for IPC requests
     */
    if (E_OK != drApiIpcCallToIPCH(&ipcClient, &ipcMsg, &ipcData))
    {
        continue;
    }

    /* Dispatch request */
    switch (drApiExtractMsgCmd(ipcMsg))
    {
    case MSG_CLOSE_TRUSTED APPLICATION:
        /**
         * Trusted Application close message
         */
        ipcMsg = MSG_CLOSE_TRUSTED APPLICATION_ACK;
        ipcData = TLAPI_OK;
        break;
    case MSG_CLOSE_DRIVER:
        /**
         * Driver close message
         */
        ipcMsg = MSG_CLOSE_DRIVER_ACK;
```

```
        ipcData = TLAPI_OK;
        break;
    case MSG_GET_DRIVER_VERSION:
        /**
         * Driver version message
         */
        ipcMsg = (message_t) TLAPI_OK;
        ipcData = DRIVER_VERSION  ;
        break;
    case MSG_RQ_EX:
        /* init tlRet value */
        tlRet = TLAPI_OK;
        /**
         * Handle incoming IPC requests via TL API.
         */
        pMarshal= (drMarshalingParam_ptr)drApiMapClientAndParams(
                                            ipcClient,
                                            ipcData);

        if (pMarshal)
        {
            /* Process the request */
            switch (pMarshal->functionId)
            {
            case FID_DR_SAMLE01:
                /**
                 * Handle Sample01 request accordingly
                 */
                break;
            default:
                /* Unknown message has been received*/
                tlRet = E_TLAPI_UNKNOWN_FUNCTION;
                break;
            }
        }

        /* Update response data */
        ipcMsg  = MSG_RS;
        ipcData = tlRet;
        break;
    default:
        /* Unknown message has been received*/
        ipcMsg  = MSG_RS;
        ipcData = E_TLAPI_DRV_UNKNOWN;
        break;
    }
}
```

The *DrApiIpcCallToIPCH* provides the Trusted Application Task ID in pIpcPeer parameter and one word of payload from the Trusted Application in the pIpcData parameter. Common usage of this parameter is to point to a data structure with encoded call parameters. A Secure Driver can then map the Trusted Application into its address space and access the data structure.

4.2.3 Accessing Trusted Application data

A Secure Driver can map a Trusted Application memory into its own memory space with Driver API memory mapping functions. This allows the Secure Driver to interpret and use data referenced using pointers in a marshalled structure.

Security Considerations

Secure Drivers must be especially careful when handling such data coming from the Trusted Application memory: the content of the Trusted Application memory may change at any time, even between two consecutive memory accesses by the Secure Driver. This means that the Secure Driver should be carefully written to avoid any security problem if this happens. If values in the buffer are security critical, the Secure Driver should always read data only once from a shared buffer and then validate it. It must not assume that data written to the buffer can be read unchanged later on.

4.2.3.1 Extended memory layout

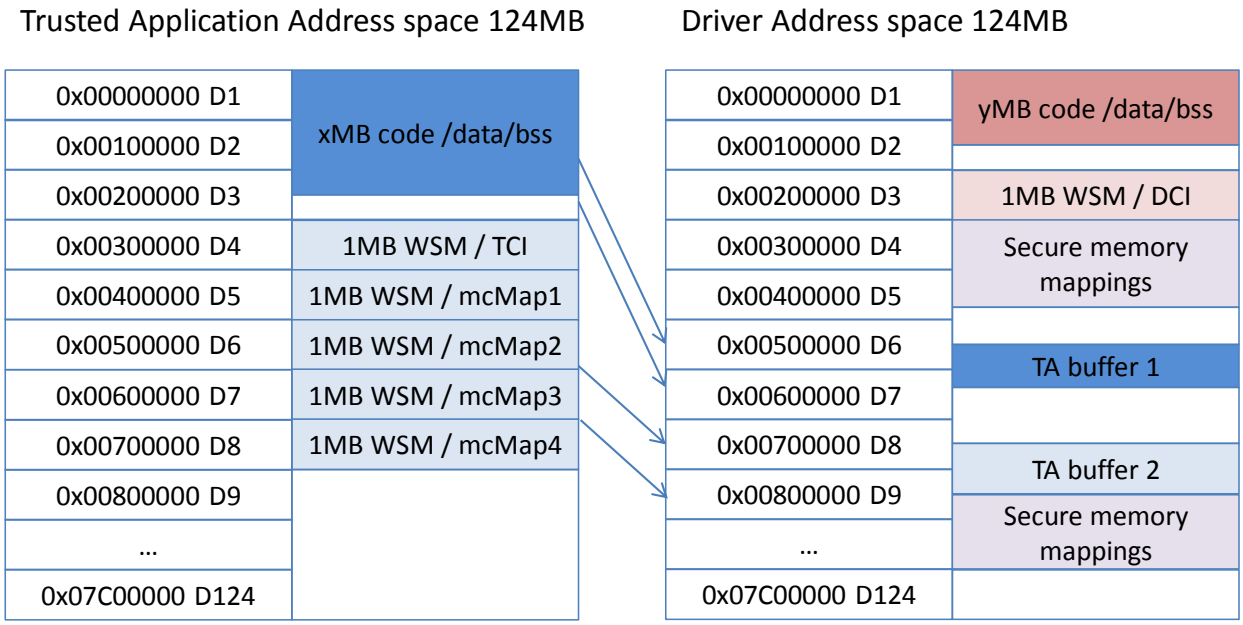


Figure 5: Mapping a Trusted Application into the Secure Driver – Extended Layout

Parameters can be mapped from the Trusted Application memory space to the Secure Driver memory space with Driver API function (extended memory layout):

```
drApiResult_t drApiMapTaskBuffer(
    const taskid_t    taskId,
    const void        *startVirtClient,
    const size_t      length,
    const uint32_t    attr,
    void              **startVirtServer
)
```

Sample usage:

```
drRet = drApiMapTaskBuffer(GET_TASK_ID(ipcClient),
                                  (addr_t )ipcData,
```

```
                                        sizeof(drMarshalingParam_ptr),
                                        MAP_READABLE|
                                        MAP_WRITABLE|
                                        MAP_ALLOW_NONSECURE,
                                        (void **)&pMarshal);
```

Trusted Application data will have different address in the Secure Driver and in the Trusted Application. This means that all Trusted Application data pointers must be converted before using them in the Secure Driver.

The flag `MAP_ALLOW_NONSECURE` can be used to allow the mapping of Non-Secure memory shared with the Normal World. This flag must be used carefully.

4.2.3.2 Legacy memory layout

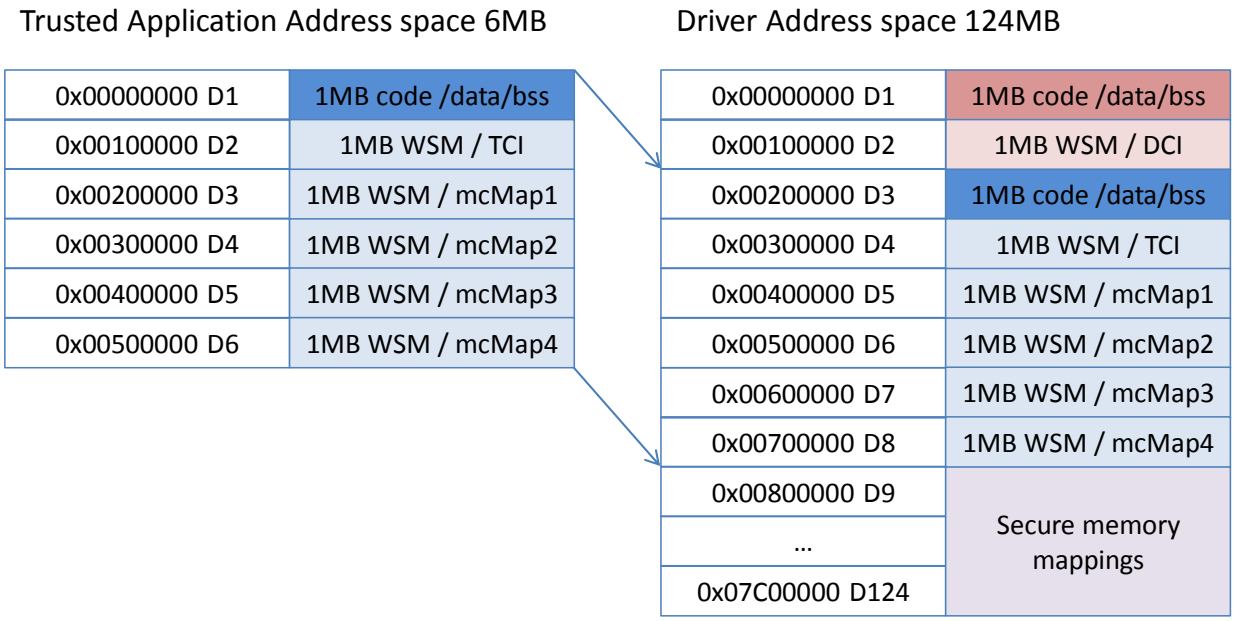


Figure 6: Mapping a Trusted Application into the Secure Driver – Legacy Layout

Parameters can be mapped from the Trusted Application memory space to the Secure Driver memory space with Driver API function:

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

Sample usage:

```
pMarshal= (drMarshalingParam_ptr)drApiMapClientAndParams(
                                        ipcClient,
                                        ipcData);
```

Trusted Application data will have different address in the Secure Driver and in the Trusted Application. This means that all Trusted Application data pointers must be converted before using them in the Secure Driver. The following Driver API function can be used to convert Trusted Application provided

pointer to Secure Driver address space. The function returns NULL, if pointer is not in correct range. Such pointer does not point to Trusted Application virtual memory space.

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

4.2.4 Unmapping Trusted Application

4.2.4.1 Extended memory layout

With this memory layout, the unmapping of the Trusted Application buffers is explicit, drivers must call `drApiUnmapTaskBuffers`() or `drApiUnmapBuffer()`.

Drivers may also choose to keep a particular TA mapping (to have a permanent mapping).

4.2.4.2 Legacy memory layout

When the Secure Driver responds to the Trusted Application using *DrApiIpcCallToIPCH,* the IPCH will automatically unmap the Trusted Application from the Secure Driver.

4.3 VIRTUAL/PHYSICAL ADDRESS TRANSLATION

Secure Drivers can translate a virtual address (from its own address space) to physical address by using the following Driver API:

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

As a result physical address is returned upon success.

4.4 HOW TO MAP PHYSICAL MEMORY

4.4.1 Extended memory layout

In order to access device registers, peripheral memory or any other memory region, Secure Drivers map physical memory into its virtual address space. The following Driver API function (extended memory layout) can be used for that purpose:

```
drApiResult_t drApiMapPhysicalBuffer(
    const uint64_t    startPhys,
    const size_t      length,
    const uint32_t    attr,
    void              **startVirtServer
)
```

Secure Drivers can do unmapping by using the following Driver API:

```
drApiResult_t drApiUnmapBuffer(const void *startVirtServer)
```

The following is a sample of how to map memory:

```
#define PHYS_MEM (uint64_t)(0x0123456789ABCDEF)
```

```
typedef struct {
   ...
   uint64_t startphys;
   void** ppDrvBuf;
   size_t size;
   uint32_t attributes;
   ...
} api_map_tst_t;


function_map_tst (...){
    void* pDrvBuf = NULL;

    api_map_tst_t api_tst;

    // init structure
    ...
    api_tst.ppDrvBuf = &pDrvBuf;
    api_tst.startphys = PHYS_MEM;
    api_tst.size = SIZE_4KB;
    api_tst.attributes = MAP_READABLE | MAP_WRITABLE | MAP_UNCACHED;

    if (E_OK != drApiMapPhysicalBuffer(
            api_tst.startphys,
            api_tst.size,
            api_tst.attributes,
            api_tst.ppDrvBuf) )
    {
        /* Mapping failed */
    }

    ...
}
```

As a result 4KB memory starting from physical address 0x0123456789ABCDEF is mapped to a virtual address found by Kinibi.

Device and memory mapping attributes are listed below:

```
/* mapping does/shall have the ability to do read access. */
#define MAP_READABLE            (1U << 0)

/* mapping does/shall have the ability to do write access. */
#define MAP_WRITABLE            (1U << 1)

/* mapping does/shall have the ability to do program execution. */
#define MAP_EXECUTABLE          (1U << 2)

/* mapping does/shall have uncached memory access. */
#define MAP_UNCACHED            (1U << 3)

/* mapping does/shall have memory mapped I/O access. */
#define MAP_IO                  (1U << 4)

/* mapping gives the ability to set Non-Secure attribute (Mutual
exclusive with MAP_EXECUTABLE). */
```

```
#define MAP_NOT_SECURE          (1U << 7)

/* mapping gives the ability to access memory with the Strongly
Ordered attribute. */
#define MAP_STRONGLY_ORDERED    (1U << 8)
```

4.4.2 Legacy memory layout

In order to access device registers, peripheral memory or any other memory region, Secure Drivers map physical memory into its virtual address space. The following Driver API function can be used for that purpose:

```
drApiResult_t drApiMapPhys(
    const addr_t      startVirt,
    const uint32_t    len,
    const addr_t      startPhys,
    const uint32_t    attr
);
```

Secure Drivers can do unmapping by using the following Driver API:

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

The following is a sample of how to map memory:

```
#define DR_VA_BUFFER (0x80000)
#define PHYS_MEM (0x12345678)


if (E_OK != drApiMapPhys(
            (page4KB_ptr) DR_VA_BUFFER,
            SIZE_4KB,
            (page4KB_ptr)(PHYS_MEM & ~(SIZE_4KB - 1)),
            MAP_READABLE | MAP_WRITABLE | MAP_UNCACHED))
    {
        /* Mapping failed */
    }
```

As a result 4KB memory starting from physical address *0x12345000* is mapped to virtual address *0x80000*.

Device and memory mapping attributes are listed below:

```
/* mapping does/shall have the ability to do read access. */
#define MAP_READABLE            (1U << 0)

/* mapping does/shall have the ability to do write access. */
#define MAP_WRITABLE            (1U << 1)

/* mapping does/shall have the ability to do program execution. */
#define MAP_EXECUTABLE          (1U << 2)

/* mapping does/shall have uncached memory access. */
#define MAP_UNCACHED            (1U << 3)
```

```
/* mapping does/shall have memory mapped I/O access. */
#define MAP_IO                  (1U << 4)

/* mapping gives the ability to set Non-Secure attribute (Mutual
exclusive with MAP_EXECUTABLE). */
#define MAP_NOT_SECURE          (1U << 7)

/* mapping gives the ability to access memory with the Strongly
Ordered attribute. */
#define MAP_STRONGLY_ORDERED    (1U << 8)
```

Secure Drivers should map devices (peripheral registers) into the designated device mappings area starting after the code, data and bss sections. Secure Drivers can map secure memory between [0x0080_0000:0x3C00_0000[. See section 3.4.2. Note that a Secure Driver has to manage its own virtual address space if it needs to map several devices or memory ranges.

4.5 HOW TO USE THREADS

When Secure Driver starts, it initially has a single thread called main thread. The main thread can start other threads by calling the following function:

```
drApiResult_t drApiStartThread(
    const threadno_t    threadNo,
    const addr_t        threadEntry,
    const stackTop_ptr  stackPointer,
    const uint32_t      priority,
    const threadno_t    localExceptionHandlerThreadNo
);
```

Since API LEVEL 8 and with 310B, the developer can use the following function to allocate a MMU-protected stack:

```
drApiResult_t  drApiStackAlloc(
     uint32_t   const stackSize,
     uint8_t  **const stackAddr
)
```

The following is a sample usage up to 310A:

```
DECLARE_STACK(drIpchStack, 2048);

/* Priority definitions */
#define EXCH_PRIORITY       MAX_PRIORITY
#define IPCH_PRIORITY       (MAX_PRIORITY-1)
…
…
/* Thread numbers */
#define DRIVER_THREAD_NO_EXCH      1
#define DRIVER_THREAD_NO_IPCH      2

…
…
    if (E_OK != drApiStartThread(
                    DRIVER_THREAD_NO_IPCH,
```

```
                    FUNC_PTR(drIpch),
                    getStackTop(drIpchStack),
                    IPCH_PRIORITY,
                    DRIVER_THREAD_NO_EXCH))
    {
        /* Starting thread failed*/
    }
```

In this particular example, exception handler thread (thread #1) is registered as local excepting handler for IPC handler thread, meaning that if IPC handler thread causes any exception, the exception handler thread will be notified. If exception is something that can be recovered, the exception handler thread can restart the thread.

The following is a sample usage from 310B:

```
// DECLARE_STACK(drIpchStack, 2048); <-- not supported anymore

/* Priority definitions */
#define EXCH_PRIORITY       MAX_PRIORITY
#define IPCH_PRIORITY       (MAX_PRIORITY-1)
…
…
/* Thread numbers */
#define DRIVER_THREAD_NO_EXCH      1
#define DRIVER_THREAD_NO_IPCH      2

uint32 t *drIpchStack;
#define DRIVER_IPCH_STACK_SIZE 4096
…
…

    if (E_OK != drApiAllocStack(
                    DRIVER_IPCH_STACK_SIZE,
                    &drIpchStack))
    {
        /* Stack allocation failed*/
    }
    if (E_OK != drApiStartThread(
                    DRIVER_THREAD_NO_IPCH,
                    FUNC_PTR(drIpch),
                    drIpchStack,
                    IPCH_PRIORITY,
                    DRIVER_THREAD_NO_EXCH))
    {
        /* Starting thread failed*/
    }
```

The stack size will be rounded up to 4KB and the stack will be allocated in virtual memory with an unmapped page before and after the stack area. The stack should be allocated once in the startup phase of the driver and shall be reused when drApiRestartThread is used.

4.6 HOW TO HANDLE EXCEPTIONS

As stated earlier, the main thread can act as exception handler. Once it is registered as local exception handling of other threads while starting them, it will receive notifications when exceptions caused by

other local threads occur. The following is a sample code that can be used to handle exceptions and restart exception causing thread:

```
/* Kernel exceptions */
#define TRAP_UNKNOWN        ( 0)    /* unknown exception. */
#define TRAP_SYSCALL        ( 1)    /* invalid syscall number. */
#define TRAP_SEGMENTATION   ( 2)    /* illegal memory access. */
#define TRAP_ALIGNMENT      ( 3)    /* misaligned memory access. */
#define TRAP_UNDEF_INSTR    ( 4)    /* undefined instruction. */
…
…
   for (;;)
    {
        /* Wait for exception */
        if ( E_OK != drApiIpcWaitForMessage(
                     &ipcPartner,
                     &mr0,
                     &mr1,
                     &mr2) )
        {
            /* Unable to receive IPC message */
            continue;
        }

        /*
            *mr0 holds threadid value of thread
            * that caused the exception
            */
        faultedThread = GET_THREADNO(mr0);

        /* Process exception */
        switch(mr1)
        {
            //-------------------------------------
            case TRAP_SEGMENTATION:
                /* Check which thread caused exception */
                switch(faultedThread)
                {
                    //-------------------------------------
                    case DRIVER_THREAD_NO_IPCH:
                        /* Update sp and ip accordingly */
                        ip = FUNC_PTR(drIpchLoop);
                        sp = getStackTop(drIpchStack);

                        /* Restart thread execution */
                        if (E_OK != drApiRestartThread (
                                        faultedThread,
                                        ip,
                                        sp))
                        {
                           /* failed */
                        }

                        break;
                    //-------------------------------------
```

```
                    default:
                        /* Unknown thread*/
                        break;
                }

                break;
            //-------------------------------------
            case TRAP_ALIGNMENT:
            case TRAP_UNDEF_INSTR:
                /**
                 * This should never happen.
                       * If it does, do the cleanup and exit gracefully
                 */
                break;
            //-------------------------------------
            default:
                /**
                 * Unknown exception occurred.
                 */
                break;
        }
    }
```

4.7 HOW TO HANDLE INTERRUPTS

In Kinibi, interrupt service routines are run in their own threads. It means that a dedicated thread needs to be started for handling ISR. This can be done from the Secure Driver main thread by calling Driver API function *drApiStartThread()*. The following is a sample code:

```
    if (E_OK != drApiStartThread(
                    DRIVER_THREAD_NO_ISRH,
                    FUNC_PTR(drIsrh),
                    getStackTop(drIsrhStack),
                    ISRH_PRIORITY,
                    DRIVER_THREAD_NO_EXCH))
    {
        /* Starting thread failed*/
    }
```

In order to be able to use interrupts they need to be attached to the interrupt handler. This can be done by calling following Driver API function:

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

In most cases, the mode parameter will be INTR_MODE_RAISING_EDGE, as interrupts usually indicate that a certain event has happened.

After attaching to an interrupt, you will also need to implement your own waiting loop for getting notified when interrupt occurs.

The following API is used to wait for an interrupt:

```
drApiResult_t drApiWaitForIntr(
```

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

When done, you can detach from a particular interrupt by calling the following Driver API function:

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

Normally interrupt handling thread is never terminated.

4.7.1 Communication between ISR thread and other local threads

As threads share the same address space there are several methods to implement synchronization and communication between threads. The method to be used should be selected based on the use case scenario the threads are to be used.

Parameters to ISR routine and back should be delivered via global variables. It is not possible to use IPC mechanism in interrupt service loop.

Sending events from ISR to other local threads can be done by using IPC signaling mechanism.

4.8 HOW TO USE SIGNALING FUNCTIONS

The following Driver API function can be used to signal en event to other drover threads:

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

A signal operation is asynchronous which means that the operation will return immediately without blocking the caller.

Receiver thread can call the following blocking function to receive signal:

```
drApiResult_t drApiIpcSigWait( void );
```

If no signal is pending the caller thread is blocked until a signal arrives.

4.9 INIT ENTRY POINT FOR DRIVERS

It is possible to define an optional _init() entry point which is called before the drMain() entry point.

```
void _init(void)
{
    // do your startup tests
}
```

This entry point can be used, for example, to perform "power-on" tests. If the power-on tests fail, the Driver can call the drApiExit() function to stop the execution.

5 USING A SECURE DRIVER

5.1 HOW TO INSTALL A SECURE DRIVER

Secure Drivers are handled like Trusted Applications and have a UUID and .tlbin file ending. Secure Drivers have to be placed in '*/data/app/mcRegistry*' folder on the device.

5.2 HOW TO LOAD A SECURE DRIVER

There are two options to load a Secure Driver:

- Secure Driver can be loaded by Kinibi normal world daemon on startup. This is done by running Kinibi daemon with '-r' option. In that case, the Secure Driver must have **.drbin* file name ending. The following is an example of the call:

  ./mcDriverDaemon -r /data/app/mcRegistry/01020304050000000000000000000000.drbin
- Secure Driver can be loaded dynamically by a TLC by simply opening a session and providing UUID of the Driver.

It is not possible for Trusted Applications to load Secure Drivers dynamically.

5.3 HOW TO CALL A SECURE DRIVER

A Trusted Applications can call a Secure Drivers through the `tlApi_callDriverEx`() function. `tlApi_callDriver()` is deprecated.

```
tlApiResult_t tlApi_callDriverEx(
      uint32_t driver_ID,
      void* payload,
      uint32_t payloadSize)

tlApiResult_t tlApi_callDriver(
      const uint32_t driver_ID,
      void* pMarParam) DEPRECATED
```

It is recommended that Secure Driver developers provide a dedicated API and library to the Trusted Applications, which can be statically linked to the Trusted Application code.

This allows hiding the calls to the Secure Driver and parameter marshalling into a functional API.

The following illustrates a sample API implementation:

```
int32_t sample_api(
    uint8_t  *src,
    uint32_t  srclen )
{
    drMarshalingParam_t marParam = {
            .functionId = FID_DR_SAMPLE,
            .payload    = {
                        .sampleData = {
                                .src     = src,
                                .dst     = dst,
```

```
                                .srclen  = srclen,
                        }
                }
        };

        return tlApi_callDriverEx(
                    SAMPLE_DR_ID,
                    &marParam,
                    sizeof(marParam));
}
```

The function `tlApi_callDriverEx()` is a blocking call that forces Trusted Applications to wait for a response from a Secure Driver. Any data that should be sent back along with the response should be communicated via buffers allocated by the Trusted Application. This buffer can be on the stack of the Trusted Application-Secure Driver library. Pointers to these buffers can be transferred to the Secure Driver in marshaling structures when the Secure Driver is called.

5.4 HOW TO DEBUG A CRASH OF A SECURE DRIVER

When a driver is crashing, it is possible to retrieve some useful information if the driver was built with the debuggable flag.

The following line must be present in the makefile

```
DRIVER_FLAGS:= 4
```

This flag must never be set for a commercial version of the Driver.

When the driver is crashing, the following traces can be retrieved from the kernel messages (dmesg)

```
<6>[  495.260097] Trustonic TEE: 101|EXCEPTION in 501, thread 0x10005, cpsr=0x60000030 [USER,TCZ]
<6>[  495.266988] Trustonic TEE: 101|cause=0x2 (TRAP_SEGMENTATION), meta=0x0
<6>[  495.273381] Trustonic TEE: 101|DFSR=0xa07 [TranslationL3,write], ADFSR=0, DFAR=0x0 [valid]
<6>[  495.281529] Trustonic TEE: 101|r0=0x00000003, r1=0x00000000,  r2=0xdeaddead,  r3=0x00007db4
<6>[  495.289742] Trustonic TEE: 101|r4=0x00000000, r5=0x00000000,  r6=0x00000000,  r7=0x00000000
<6>[  495.298058] Trustonic TEE: 101|r8=0x00000000, r9=0x00000000, r10=0x00000000, r11=0x00000000
<6>[  495.306291] Trustonic TEE: 101|r12=0x03d04fbf,sp=0x00007da8,  lr=0x00001e5d,  pc=0x000010e6
<6>[  495.314601] Trustonic TEE: 101|UUID=02040000000000000000000000000000
```

5.5 KINIBI HALTED

When Kinibi detects that a driver crashed, Kinibi halts itself and prevents the execution of the other drivers and Trusted Applications.

The following traces can be found in the logcat in order to identify the halt reason:

```
10-28 21:27:09.965 E/McDaemon( 1956):   ****************************
10-28 21:27:09.965 E/McDaemon( 1956):   *** ERROR: Trustonic TEE halted. Status dump:
10-28 21:27:09.965 E/McDaemon( 1956):   *** Detected in logCrashDump/253()
10-28 21:27:09.965 E/McDaemon( 1956):   ****************************
10-28 21:27:09.965 I/McDaemon( 1956):   flags               = 0x80000401
10-28 21:27:09.965 I/McDaemon( 1956):   haltCode            = 0x00000002
10-28 21:27:09.965 I/McDaemon( 1956):   haltIp              = 0x03f07e74
10-28 21:27:09.965 I/McDaemon( 1956):   faultRec.cnt        = 0x00000006
10-28 21:27:09.965 I/McDaemon( 1956):   faultRec.cause      = 0x00000002
10-28 21:27:09.965 I/McDaemon( 1956):   faultRec.meta       = 0x00000000
10-28 21:27:09.965 I/McDaemon( 1956):   faultRec.thread     = 0x00010005
10-28 21:27:09.965 I/McDaemon( 1956):   faultRec.ip         = 0x000010e6
10-28 21:27:09.965 I/McDaemon( 1956):   faultRec.sp         = 0x00007da8
10-28 21:27:09.965 I/McDaemon( 1956):   faultRec.arch.dfsr  = 0x00000a07
10-28 21:27:09.965 I/McDaemon( 1956):   faultRec.arch.adfsr = 0x00000000
10-28 21:27:09.965 I/McDaemon( 1956):   faultRec.arch.dfar  = 0x00000000
10-28 21:27:09.965 I/McDaemon( 1956):   faultRec.arch.ifsr  = 0x00000207
```

```
10-28 21:27:09.965 I/McDaemon( 1956):   faultRec.arch.aifsr = 0x00000000
10-28 21:27:09.965 I/McDaemon( 1956):   faultRec.arch.ifar  = 0x00000000
10-28 21:27:09.965 I/McDaemon( 1956):   mcData.flags        = 0x00000001
10-28 21:27:09.965 I/McDaemon( 1956):   mcExcep.partner     = 0xffffffff
10-28 21:27:09.965 I/McDaemon( 1956):   mcExcep.peer        = 0x00010005
10-28 21:27:09.965 I/McDaemon( 1956):   mcExcep.cause       = 0x00000002
10-28 21:27:09.965 I/McDaemon( 1956):   mcExcep.uuid        = 0x02040000000000000000000000000000
10-28 21:27:09.965 I/McDaemon( 1956):   mcExcep.meta        = 0x00000000
10-28 21:27:09.965 E/McDaemon( 1956):   ****************************
```

6 Trustonic DDK

Trustonic DDK is the Driver Development Kit for Kinibi. It provides documentation, libraries and sample code to develop Secure Drivers.

6.1 DRIVER API

A Secure Driver has specific set of functions available in Driver API. Files that are included in the API are listed in the table. Please see the following header files for more information about the APIs.

Table 1: DrApi header files and libraries

File name	Details
DrApi.lib	Library file that implements the Driver API
DrEntry.lib	Library file that implements runtime environment for each Driver
drStd.h	This is a wrapper for standard header files; it also specifies macros to declare stack area.
DrApi.h	Include file to take Driver API library into use
DrApiCommon.h	Include file defining data types
DrApiError.h	Error codes to be used in Trusted Application-Driver API
DrApiFastCalls.h	Include file for FastCalls handling
DrApiHeap.h	Include file for Dynamic Memory Allocation API
DrApiIpcMsg.h	Include file for Inter Process Communication API
DrApiLogging.h	Include file for Logging API
DrApiMcSystem.h	Include file for System API
DrApiMm.h	Include file for Memory Management API
DrApiMmEx.h	Include file for Extended Memory Management API
DrApiThread.h	Include file for Thread and Interrupt API
version.h	Version of Driver API including major and minor number

6.2 SECURE DRIVER TEMPLATE

DrTemplate is a template of Secure Driver that shows:

- the usage of exception handler thread
- the usage of IPC handler thread
- the usage of DCI handler thread
- sample implementation of Driver APIs
- sample session management

6.2.1 DrTemplate structure

The Secure Driver template consists of the following components

- Exception handler thread (main thread):

  Responsible for handling exceptions caused by IPC and DCI handler threads. Exception handler thread number is #1 and it has the highest priority than IPC and DCI handler threads.

- IPC handler thread:

  Responsible for handling IPC messages coming from RTM IPCH and also from other Trusted Applications that use the APIs provided by the template. IPC handler thread number is #2. If IPC handler thread receives Trusted Application and Driver close acknowledgement request from RTM IPCH, Secure Driver needs to do cleanup if necessary and acknowledge to IPCH.

- DCI handler thread:

  Responsible for handling notifications that arrive from normal world DCI (Driver Control Interface) handler. This is exactly same as how Trusted Applications communicate with normal world via TCI. Data exchange between two worlds is possible via DCI buffer. It is possible to disable DCI handler during compilation time by setting 'USE_DCI_HANDLER' to 'NO'

- Trusted Application API:

  Trusted Application API provides interface to other Trusted Applications to request services from the Secure Driver. It is possible to disable Trusted Application API interface during compilation time by setting 'USE_TL_API' to 'NO'

- Session Management:

  Session management is responsible for keeping session registry and handling session registry data. Session management implementation is not a must in a Secure Driver implementation. Driver can also handle Trusted Applications without maintaining sessions. This will require that each execute requests will be handled at once. Session management also requires that IPC data sent by a Trusted Application is copied by the Secure Driver for later use. If session-less mode is used, Trusted Application memory can be mapped by the Secure Driver and then the Secure Driver directly accesses data.

6.2.2 What needs to be updated

You may want to:

- update file names, makefile.mk content
- update variable & structure names/members
- add/remove new structures, etc.
- add/remove new functions for handling DCI and IPC messages
- add cleanup functionality in the exception handler
- add additional threads, for example ISR handler thread
- General definitions

- General Driver definitions can be found in '*drCommon.h*'. The definitions include thread numbers and priorities, various macros.

- Trusted Application API
  - if Trusted Application API is offered, update '*drApiMarshal.h*', '*tlDriverApi.c*' and '*tlDriverApi.h*' for defining new marshalling parameters, new function IDs, new APIs, etc..
  - You will also need to update '*drIpcHandler.c*' for handling incoming requests. As default, Trusted Application APIs are enabled, but this can be updated in makefile.mk
- DCI handler
  - If you would like to add new data structure definitions in DCI buffer, you need to modify '*dci.h*' and '*drTemplate_Api.h*'.
  - You will also need to modify '*drDciHandler.c*' for handling incoming DCI messages.
- IPC handler thread
  - Update incoming IPC message handling for functions. This may also require updates in the session management part if session management is in use.
  - You will also need to handle Trusted Application and Driver close acknowledgement requests accordingly in drIpcHandler.c and do necessary cleanup before sending any acknowledgement to IPCH
  - Update *SAMPLE_DR_ID* accordingly in '*drApiMarshal.h*'. This must be the same Driver ID value as in makefile.mk
- Exception handling
  - Update '*drExcHandler.c*' to do the cleanup before stopping threads in case of '*TRAP_UNDEF_INSTR*'.
  - After restarting IPC or DCI handler thread, you need to respond to Trusted Application or notify normal world and indicate with a valid error code if there is an outstanding request.
- Main entry point
  - You need to implement initialization functionality if your Driver needs certain initialization.
  - You can update '*doInitialization(void)*' in '*drMain.c*' with specific initialization code. For example certain HW initialization needs to be taken case before launching Driver threads or if you would like to do physical memory mapping, etc.., this is the function where this can be handled
- Session Management
  - Session management can be optional
  - Session management is implemented in '*drSmgmt.c*'. Both function and data definitions can be found in *'drSmgmt.h*'.
  - If you would like to add new session states, you can update *'sessionState_t*.
  - If new session registry data members are required, you can update *'drSessionReg_t*' structure definition.
  - If needed, you can update both *'drSmgmt.c'* and '*drSmgmt.h*' for adding new functions for handling additional session management operations.
- makefile.mk
  - Update *DRIVER_UUID* accordingly
  - Update file names accordingly if you rename source files
  - *Update USE_DCI_HANDLER* and *USE_TL_API* default values accordingly

6.3 EXAMPLES

6.3.1 Async example

This is an example of asynchronous calls in Kinibi: drApiIpcSignal function invoked in IPC thread of the Secure Driver sends a signal, which is caught by drApiIpcSigWait function in Interrupt thread, Interrupt

thread runs a hardware or software timer that elapses and DrApiNotifyClient function is invoked to awake a calling Trusted Applications. The sample also contains platform dependent timer implementation, which uses assembler code, for Versatile and Arndale platforms.

6.3.2 Rot13 example

This example shows a complete calling path from the Application, Trusted Application Connector, Trusted Application, Secure Driver (without hardware access use).

6.4 BUILDING A SECURE DRIVER

In general the following command specifying chosen TOOLCHAIN, PLATFORM and MODE is used to build a driver:

```
TOOLCHAIN=GNU MODE=Debug PLATFORM=ARM_V7A_STD ./Locals/Build/build.sh
```

By default the following setting are used:

PLATFORM : ARM_V7A_STD

TOOLCHAIN : ARM

MODE : Debug

6.4.1 Extended memory layout

A driver with the extended memory layout can support any type of Trusted Application (with legacy or extended memory layout).

In order to use the extended memory layout, the following line must be added to the driver makefile:

```
TBASE_API_LEVEL := 5 (or higher)
```

6.4.1.1 Mapping APIs

With this layout, only the following mapping functions can be used:

- drApiMapTaskBuffer()
- drApiMapPhysicalBuffer()
- drApiUnmapBuffer()
- drApiUnmapTaskBuffers()

The following functions cannot be used:

- drApiAddrTranslateAndCheck
- drApiMapClientAndParams
- drApiMapPhys/drApiMapPhys64
- drApiUnmap
- drApiMapPhysPage4KB / drApiMapPhysPage4KB64
- drApiMapPhysPage4KBWithHardware/drApiMapPhysPage4KBWithHardware64
- drApiUnmapPage4KB

6.4.1.2 Heap

The main heap of the driver is optional.

In order to declare a main heap, the following lines must be added to the makefile:

```
HEAP_SIZE_INIT := 4096 (for example)
HEAP_SIZE_MAX := 8192 (optional)
```

6.4.2 Floating point support

Floating point can be used in secure drivers with the same conditions as for Trusted Applications.

In order to use hardware floating point, the following lines must be added to the driver makefile:

```
HW_FLOATING_POINT := Y
TBASE_API_LEVEL := 5 (or higher)
```