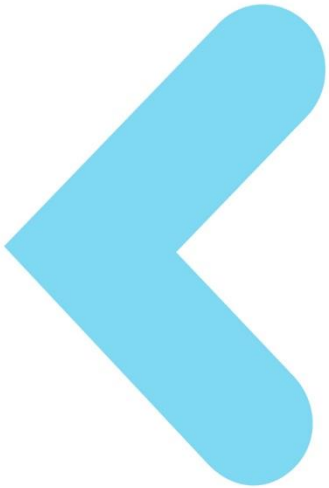
TRUSTONIC



Kinibi TEE
System Debug Guide

PREFACE

VERSION HISTORY

Version	Date	Modification
0.1	2018, April the 4th	Original draft
1.0	2018, May the 3rd	First complete version, covering ClientApp and TrustedApp errors, debug mechanism from SWd, profiling tool and post-mortem analysis.
1.1	2018, June the 6th	Few more debug info added (debug log in case of invalid TA signature, TA disassemble info, TEE memory mapping, post-mortem investigation sequence)
1.2	2018, October the 1st	Updating traces/info to match Kinibi-410 evolutions (CPU ID in logs, few more tee-ps features).

TABLE OF CONTENTS

1 INTRODUCTION

This document is regrouping a set of notes around the debug of Trusted Application, Secure World driver and also covering the system integration support in case of TEE unrecoverable crash. The information presented below are coming from different Trustonic documentations or feedback from the field.

The target audience is for system integrators or engineers already aware of Kinibi development environment.

The first thing to introduce is the different debug mechanisms used to Kinibi:

- The Android/Linux `logcat` service for all the error messages reported by the userspace components (mcDriverDaemon or libMcClient)
- The kernel Linux `dmesg` service for the error messages raised by the Kinibi Linux driver and all the messages from the Secure World (TEE itself but also TAs and SWd drivers).
- The Linux `DebugFS` files to allow global view of low level TEE exchanges with the SWd (feedback from TEE Linux kernel context).
- A specific Kinibi debug tools, `TeePS`, to provide some none sensitive information on SWd context (TAs currently running, few memory info…)
- And finally, for development phases, Trustonic is also providing a Debug image for the TEE itself, with many system logs enabled, allowing to get a deep feedback from Secure Worlds activities.

As a general comment to start any real investigation on an issue or to allow Trustonic to perform efficient support, it is mandatory get at least:

- The exact product version used (for example, `t-base-Taget-Android-311C-V004`) as feature set or fixes are different from one to another.
- The 2 Linux system logs, `logcat` and `dmesg`.

2 CONFIRM TEE RUNNING

The very first operations to do in case of new integration or TEE crash is to verify overall state of TEE components:

Check TEE mcDriverDaemon still running (access to the TEE registry to load TA binaries)?

```
adb_shell:/ # ps | grep mcDriverDaemon
drmrgp  1922  1  14556  3556  futex_wait 7befa79cb0 S /system/bin/mcDriverDaemon
adb_shell:/ #
```

Check TEE Linux driver well loaded into the Linux kernel (global TA communication flow to the SWd)?

```
adb_shell:/ # ls -la /dev/mobi*
crw------- 1 drmrgp  drmrgp 241,   0 2017-01-01 12:01 /dev/mobicore
crw-rw---- 1 drmrgp  drmrgp 241,   1 2017-01-01 12:01 /dev/mobicore-user
adb_shell:/ #
```

And finally use the `tlcInfo` basic tool, provided in Kinibi TEE package under `\t-base-dev-kit\Tools\Info\arm64-v8a\Debug` (confirm that a minimal communication with the SWd is possible) :

```
adb_shell:/data # ./misc/mcRegistry/tlcInfo
Copyright (c) Trustonic Limited 2013-2015.
t-base-Android-400A-V012-20170726_155303_34988_66846, Jul 26 2017, 16:25:40.

mcGetMobiCoreVersion:
productId        = t-base-EXYNOS64-Android-400A-V012-20170726_155303_34988_66846
versionMci       = 0x00010006
versionSo        = 0x00020002
versionMclf      = 0x00020005
versionContainer = 0x00020001
versionMcConfig  = 0x00000003
versionTlApi     = 0x00010014
versionDrApi     = 0x00010004
versionCmp       = 0x00060003
```

```
CMP GetSuid:
SUID             = 0x02000000005088E713A5984EF7070000
SUID.SiP         = 0x00000002
SUID.SoC         = 0xE7885000
SUID.CSN         = 0x13A5984EF7070000
CMP GetVersion:
productId        = t-base-EXYNOS64-Android-400A-V012-20170726_155303_34988_66846
versionMci       = 0x00010006
versionSo        = 0x00020002
versionMclf      = 0x00020005
versionContainer = 0x00020001
versionMcConfig  = 0x00000003
versionTlApi     = 0x00010014
versionDrApi     = 0x00010004
versionCmp       = 0x00030000
adb_shell:/data #
```

In the case of one of these commands does not return expected information it use useless to continue. It means that one of the mandatory component is not present or fatally crashed.

If missing mcDriverDaemon or Linux TEE driver, Integration need to be fixed in order to start/Load them at platform boot.

In the case of issue with `tlcInfo`, device need to be rebooted first to confirm TEE can boot correctly. If after reboot `tlcInfo` is working correctly, then next sections of this documentation should help to localize fatal crash root cause. If, even after a reboot SWd is not responding then it means TEE has mostly not booted correctly (platform integration has also to be fixed, mostly relying on TEE debug image to get feedback on blocking point).

3 DEBUGGING ERRORS FROM CLIENT APPLICATIONS

The Client Applications are standard OS applications and thus all usual debug tools of the OS are perfectly applicable (printf(), logs in logcat, usage of application debugger like GDB…). For deeper information, a complete section is provided by Trustonic in *Kinibi_Developers_Guide.pdf*.

This section will more focus on the potential errors returned by the TEE User Space interfaces, the **legacy Trustonic Client API** and the **GP Client API**.

Extract from file `\t-base-dev-kit\t-sdk\TlcSdk\Public\MobiCoreDriverApi.h` for error codes returned by **Legacy Trustonic Client API**:

```
#define MC_DRV_OK                 0x00000000 /**< Function call succeeded. */
#define MC_DRV_NO_NOTIFICATION    0x00000001 /**< No notification available. */
#define MC_DRV_ERR_NOTIFICATION   0x00000002 /**< Error during notification on communication level. */
#define MC_DRV_ERR_NOT_IMPLEMENTED  0x00000003 /**< Function not implemented. */
#define MC_DRV_ERR_OUT_OF_RESOURCES 0x00000004 /**< No more resources available. */
#define MC_DRV_ERR_INIT        0x00000005 /**< Driver initialization failed. */
#define MC_DRV_ERR_UNKNOWN     0x00000006 /**< Unknown error. */
…
```

Extract from file `\t-base-dev-kit\t-sdk\TlcSdk\Public\GP\tee_client_error.h` for error codes returned by **GP Client API:**

```
#define TEEC_ERROR_GENERIC                  ((TEEC_Result)0xFFFF0000)
#define TEEC_ERROR_ACCESS_DENIED            ((TEEC_Result)0xFFFF0001)
#define TEEC_ERROR_CANCEL                   ((TEEC_Result)0xFFFF0002)
#define TEEC_ERROR_ACCESS_CONFLICT          ((TEEC_Result)0xFFFF0003)
#define TEEC_ERROR_EXCESS_DATA              ((TEEC_Result)0xFFFF0004)
#define TEEC_ERROR_BAD_FORMAT               ((TEEC_Result)0xFFFF0005)
…
```

In case of errors returned by these APIs, if the error code or comments from the header file or the API documentation are not enough, the right way to analyze is to start looking at `logcat` to check if the root cause is generated by one of the Normal World components (Trusted App binary not found, Session already closed by someone else…)

Below are typical valid logs from `mcDriverDaemon` and a quick filter on pattern "**Tee**" should highlight the interesting part:

```
[7.412902][4: mcDriverDaemon] 01-01 13:47:58.446  2511  2511 I TeeMcDaemonRegistry search paths:
[./MobiCoreDriverDaemon.cpp:204]
[7.412959][4: mcDriverDaemon] 01-01 13:47:58.447  2511  2511 I TeeMcDaemon /data/vendor/misc/mcRegistry
[./MobiCoreDriverDaemon.cpp:206]
[7.412992][4: mcDriverDaemon] 01-01 13:47:58.447  2511  2511 I TeeMcDaemon  /vendor/app/mcRegistry
[./MobiCoreDriverDaemon.cpp:206]
[7.413022][4: mcDriverDaemon] 01-01 13:47:58.447  2511  2511 I TeeMcDaemon  /system/app/mcRegistry
[./MobiCoreDriverDaemon.cpp:206]
[7.413064][4: mcDriverDaemon] 01-01 13:47:58.447  2511  2511 I TeeMcDaemon Initialise Secure World
[./MobiCoreDriverDaemon.cpp:236]
 [7.416530][0: mcDriverDaemon] 01-01 13:47:58.450  2511  2511 I TeeMcDaemon driver open [./driver.cpp:110]
[7.416605][0: mcDriverDaemon] 01-01 13:47:58.450  2511  2511 I TeeMcDaemon Kinibi access granted by the driver [./common.cpp:197]
 [7.417074][0: mcDriverDaemon] 01-01 13:47:58.451  2511  2511 I TeeMcDaemon TEE is ready, version: t-base-Exynos-Android-410a-
v001-20180329_190510_46723_76543
…
```

However, a case of a Client App trying to open a Trusted App not found in the TEE registry will output this type of error message

```
[7.424982][6: McDaemon.SWd] 01-01 13:47:58.459  2511  2516 W
TeeMcDaemon Cannot open trustlet
data/vendor/misc/mcRegistry/07050501000000000000000000000020.tlbin (No such file or directory) [./SecureWorld.cpp:195]
```

Please note that Trustonic is providing in TEE package Release and Debug versions of these components. If the release version outputs in logcat only minimal error messages, the Debug version will be much more verbose and could be helpful during development.

In case nothing suspicious/strange found in the `logcat`, the next step will consist in looking at the Linux kernel logs with `dmesg`. Ideally using the TEE debug binary to get a verbose feedback, topic is covered in next sections.

4 DEBUGGING ERRORS FROM SECURE WORLD

Kinibi is a secure runtime environment and thus there are limited debugging options on the target platforms. In fact, debugging the SWd may not be allowed at all on end user platforms.

Hint: Actively debugging Trusted Applications bypasses security restrictions enforced by Kinibi and TrustZone. As a consequence, a debug-able system can no longer be considered as a secure environment. Therefore it is strongly recommended that any involved background system is aware that it is dealing with a non-secure development platform.

Once accepted the security related limitations, all the Secure World logs from the TEE or any components running on top of it are displayed in the Linux kernel messaging service, `dmesg`.

If by default the **TEE Release** image will only output the absolute minimal messages, the **TEE Debug** binary is much more verbose and will print error messages for any problem raised inside Secure World. During development and debug Trustonic is highly recommending using it and only switch to **TEE Release** once overall use cases stables.

4.1 ERRORS FROM TRUSTED APPLICATIONS

In case of errors during the development of Trusted Application, Kinibi is offering several debug mechanisms like usual logging APIs or specific build options (`TRUSTLET_FLAGS/TA_FLAGS := 4`) to allow a Trusted App to print feedback or dump status in case of unexpected shutdown (TA crash). For deeper information, a complete section is provided by Trustonic in *Kinibi_Developers_Guide.pdf*.

It is normally impossible for a Trusted App to generate a crash of the TEE. Normally any invalid operation should either generate a clear error returned by one of the Kinib SWd internal API, the **Legacy Trustlet API**, the **GP internal API** or in the worst case the TEE will take decision to halt cleanly this task.

Extract from file `\t-base-dev-kit\t-sdk\TlSdk\Public\MobiCore\inc\TlApi \TlApiError.h` for error codes returned by **Legacy Trustlet API**:

```
#define TLAPI_OK                  0x00000000   /**< Returns successful. */
#define TLAPI_PENDING             0x00000001   /** drv response is not available yet. */
#define E_TLAPI_NOT_IMPLEMENTED   0x00000101   /**< Function not yet implemented. */
#define E_TLAPI_UNKNOWN           0x00000102   /**< Unknown error during TlApi usage. */
#define E_TLAPI_UNKNOWN_FUNCTION  0x00000104   /**< Function not known. */
#define E_TLAPI_INVALID_INPUT     0x00000105   /**< Input data is invalid. */
#define E_TLAPI_INVALID_RANGE     0x00000106   /**< If address/pointer. */
#define E_TLAPI_BUFFER_TOO_SMALL  0x00000107   /**< A buffer is too small. */
…
```

Extract from file `\t-base-dev-kit\t-sdk\TlSdk\Public\GPD_TEE_Internal_API \tee_error.h` for error codes returned by **GP internal API**:

```
#define TEE_SUCCESS                          ((TEE_Result)0x00000000)
#define TEE_ERROR_CORRUPT_OBJECT             ((TEE_Result)0xF0100001)
#define TEE_ERROR_CORRUPT_OBJECT_2           ((TEE_Result)0xF0100002)
#define TEE_ERROR_STORAGE_NOT_AVAILABLE      ((TEE_Result)0xF0100003)
#define TEE_ERROR_STORAGE_NOT_AVAILABLE_2    ((TEE_Result)0xF0100004)
#define TEE_ERROR_GENERIC                    ((TEE_Result)0xFFFF0000)
#define TEE_ERROR_ACCESS_DENIED              ((TEE_Result)0xFFFF0001)
…
```

In case of errors returned by these APIs, if the error code or comments from the header file or the API documentation are not enough, the easiest way to continue the investigation is to rely on **TEE Debug** image and look at specific TEE outputs in Linux Kernel `dmesg`.

4.2 USING KINIBI TEE DEBUG IMAGE

During development and debug phases, using Kinibi **TEE Debug** image is extremely helpful as TEE system components will print "user friendly" helper messages to feedback status of the Secure World.

A quick filtering in `dmesg` on pattern "**`Trustonic TEE`**" will provide you the interesting part:

```
[8.810966] [4: tee_log:  868] [c4] Trustonic TEE: mtk(7)|*** Trustonic TEE MTK, Build: Sep 10 2018, 10:27:56 ***
[8.810986] [4: tee_log:  868] [c4] Trustonic TEE: mtk(7)|MTK: t-base-EXYNOS-Android-410a-V003-20180910_102754_84632
[8.811003] [4: tee_log:  868] [c4] Trustonic TEE: mtk(7)|MTK: Build for AARCH32, EXYNOS64_STD
[8.811027] [4: tee_log:  868] [c4] Trustonic TEE: mtk(7)|MTK:   MIDR=0x410fd034 [ARM, ARMv8-A, Cortex-A53, r0p4]
…
[9.463079] [7: tee_log:  868] [c7] Trustonic TEE: 104(5)|MSH  EXCH stack max usage: 500 of 1024 byte
[9.463111] [7: tee_log:  868] [c7] Trustonic TEE: 104(5)|MSH  SIQH stack max usage: 324 of 1024 byte
[9.463132] [7: tee_log:  868] [c7] Trustonic TEE: 104(5)|MSH  IPCH stack max usage: 936 of 2100 byte
[9.463153] [7: tee_log:  868] [c7] Trustonic TEE: 104(5)|MSH  MSH stack max usage: 872 of 7168 byte
[9.463174] [7: tee_log:  868] [c7] Trustonic TEE: 104(5)|MSH  LOADH stack max usage: 448 of 7168 byte
[9.463191] [7: tee_log:  868] [c7] Trustonic TEE: 104(5)|MSH  Active sessions:
[9.463213] [7: tee_log:  868] [c7] Trustonic TEE: 104(5)|MSH  201: Driver, ID=0x104/260
[9.463250] [7: tee_log:  868] [c7] Trustonic TEE: 104(5)|MSH   SPID=0xfffffffe, UUID=07050500-0000-0000-0000-000000000020
[9.463272] [7: tee_log:  868] [c7] Trustonic TEE: 104(5)|MSH   State=Active, flags=0x0000001a, max instances=1 ...
..
[28.303284] [7: tee_log:  868] [c7] Trustonic TEE: 701(5)|KeyCtrl: tlMain(2404):
[28.303317] [7: tee_log:  868] [c7] Trustonic TEE: 701(5)|KeyCtrl: Got a command: 0x00000A01
[28.303317] [7: tee_log:  868] [c7] Trustonic TEE: 701(5)|KeyCtrl: Return: 0x00000000
…
```

Format of the Kinibi Debug image log are the following:

- `[8.810966] [4: tee_log:  868] [c4],` usual pattern added by Linux kernel for any kernel print.
- `Trustonic TEE`: template added by Kinibi kernel for any SWd logs
- `701`, is the TEE Process/Thread ID based on pattern ([2 Bytes PiD][2 Bytes Thread iD])
- `(5)`, Added since Kinibi-410 and early SMP support, is the CPU running the code/doing the print in SWd at that time.

- `KeyCtrl`, will be the "service tag name", defined statically for any components running in the TEE.
- Remaining will be the log itself as implemented in the service…

4.2.1 About Process/Thread ID in Kinibi

```
[22.6988] [1: kworker/1:0: 18] Trustonic TEE: 104(5)|MSH  301: Driver, ID=0x104/260
```

Any TA or SWd Driver started within Kinibi will have a Process ID and Thread ID assigned to it. It's important to well understand this part because logs or Kinibi TRM tasks dump are based on these values instead of TA UUIDs.
The very firsts processes started in the system are usually Trustonic TEE Core services (tagged with pattern 101/102/103/104 for the Kinibi RTM, 201/202/203 for the Kinibi Crypto Driver…). Then TAs have only 1 thread per process where SWd Drivers will manage several (at least 3 between the main thread, the IPC thread and the Exception thread)

Kinibi Debug image or using TeePs tool would provide an easy way to get mapping between TA/SWd Driver UUID and ProcessID/Thread ID as used by Kinibi TEE Kernel.

4.2.2 Kinibi TEE Standard Error Messages

Below are logs extracted from a Kinibi Debug image with few words to explain each situation and allow interpretation.

To ease the reading, the Linux kernel display tag (`[22.698] [1: kworker/1:0: 18]`) has been removed and focusing only on text coming from SWd.

Usual OpenSession logs

```
Trustonic TEE: 104|MSH mshHandle_OPEN_SESSION(): starting 07010000-…-000000000000
Trustonic TEE: 104|MSH  Number of Hash public key : 1
Trustonic TEE: 104|MSH registerSession(): started 801, serviceType=System TA, driverId=0
```

Displaying openSession of a TA, providing mapping between UUID and PiD (Process 8, Thread 1)

```
Trustonic TEE: 104|MSH Dumping RTM state:
Trustonic TEE: 104|MSH  Active sessions:
Trustonic TEE: 104|MSH  201: Driver, ID=0x150/336
```

```
Trustonic TEE: 104|MSH    SPID=0x00000000, UUID=02010000-0000-0000-0000-000000005000
Trustonic TEE: 104|MSH    State=Active, flags=0x0000000b, max instances=1
Trustonic TEE: 104|MSH   701: System TA
Trustonic TEE: 104|MSH    SPID=0xfffffffe, UUID=07060000-0000-0000-0000-00000000004d
Trustonic TEE: 104|MSH    State=Active, flags=0x00000008, max instances=16
Trustonic TEE: 104|MSH   801: System TA
Trustonic TEE: 104|MSH    SPID=0xfffffffe, UUID=07010000-0000-0000-0000-000000000000
Trustonic TEE: 104|MSH    State=Active, flags=0x00000000, max instances=1
Trustonic TEE: 104|MSH   Page pool-DDR usage: 137 of 6772
```

Each openSession is also going with a dump of already on-going sessions (PiD/Service type/UUID) and a sum-up of used system memory.

```
Trustonic TEE: 801|TlCm: Starting, 3.6, Mar 9 2015,
```

And finally, once TA loaded, "real" logs printed by the TA itself (we find again the TA PiD in the log pre-fix)

Usual CloseSession logs

```
Trustonic TEE: 104|MSH MCP CLOSE(): Closing 801
Trustonic TEE: 104|MSH  Terminating threadid=801
Trustonic TEE: 101|EXC EXCEPTION from 0xffffffff with 0x00000008, 0x00000012, 0xfffffffe
Trustonic TEE: 101|EXC TERMINATION: task 0x8, exitCode=0xfffffffe, UUID=07010000-…0000
Trustonic TEE: 101|EXC task 0x8 terminated, session exit code=0xffffffff
Trustonic TEE: 101|EXC RTM state when exception happened:
Trustonic TEE: 101|EXC  EXCH stack max usage: 424 of 768 byte
Trustonic TEE: 101|EXC  SIQH stack max usage: 400 of 768 byte
Trustonic TEE: 101|EXC  IPCH stack max usage: 520 of 1868 byte
Trustonic TEE: 101|EXC  MSH stack max usage: 2072 of 2304 byte
Trustonic TEE: 101|EXC  LOADH stack max usage: 88 of 2048 byte
Trustonic TEE: 101|EXC  Active sessions:
Trustonic TEE: 101|EXC  201: Driver, ID=0x150/336
Trustonic TEE: 101|EXC   SPID=0x00000000, UUID=02010000-0000-0000-0000-000000005000
Trustonic TEE: 101|EXC   State=Active, flags=0x0000000b, max instances=1
Trustonic TEE: 101|EXC  301: Driver, ID=0x104/260
Trustonic TEE: 101|EXC   SPID=0x00000000, UUID=07050500-0000-0000-0000-000000000020
Trustonic TEE: 101|EXC   State=Active, flags=0x0000000a, max instances=1
Trustonic TEE: 101|EXC  401: System TA
Trustonic TEE: 101|EXC   SPID=0xfffffffe, UUID=07050501-0000-0000-0000-000000000020
Trustonic TEE: 101|EXC   State=Active, flags=0x00000008, max instances=1
Trustonic TEE: 101|EXC  Page pool-DDR usage: 135 of 6772
Trustonic TEE: 101|EXC mark sesssion dead for task 0x8, session exit code 0xffffffff
```

In the closeSession logs, we do again the matching with the PiD. We will also get an update on overall sessions/memory available in the system (similar status than the one provided in the openSession).

```
Trustonic TEE: 104|MSH  send MSG_CLOSE_TRUSTLET sId=801 to IPCH 00010008 00000801
Trustonic TEE: 103|IPC closeEvent_enqueue(): closingThreadId 0x0108, srcThreadId 0x0401
Trustonic TEE: 103|IPC closeEvent_broadcast():closingThreadId 0x0108, srcThreadId 0x0401
Trustonic TEE: 103|IPC closeEvent_broadcast(): notify active server 0x00020002
Trustonic TEE: 103|IPC closeEvent_broadcast(): notify active server 0x00010003
Trustonic TEE: 103|IPC closeEvent_broadcast(): notify active server 0x00020005
Trustonic TEE: 103|IPC closeEvent_broadcast(): notify active server 0x00020006
Trustonic TEE: 103|IPC closeEvent_finishCurrent(): closingThreadId 0x0108,
                                                   srcThreadId 0x0401
Trustonic TEE: 103|IPC closeEvent_finishCurrent(): send MSG_CLOSE_TRUSTLET_ACK 0x00010008
Trustonic TEE: 104|MSH  received MSG_CLOSE_TRUSTLET_ACK 00010008 00000801
Trustonic TEE: 104|MSH releaseTaskViaIpch(): send MSG_RELEASE_DRAGON_TASK to IPCH,
                                             task ID=0x8
Trustonic TEE: 104|MSH ipchDeregisterClient(): thread 0x00010008, SID 801
```

Interesting part in the closeSession log is the notification loop to "servers" or SWd drivers. In Kinibi each "server" must acknowledge the end of another service. Not sending the acknowledge signal could lead to a system deadlock. This log is particularly useful to detect this type of development issue.

Failing OpenSession due to invalid TA signing key

```
Trustonic TEE: 104|MSH mshHandle_OPEN_SESSION(): starting 00000000-1111-1111-1111-
                      111100000000
Trustonic TEE: 104|MSH  Number of Hash public key : 1
Trustonic TEE: 104|MSH   Public key hash missmatch;Locals/Code/MC/RTM/LL/llCrypto.c:444
Trustonic TEE: 104|MSH   Throw exception 0x40b00010 at;
                         Locals/Code/MC/RTM/LL/llCrypto.c:444
Trustonic TEE: 104|MSH smgntOpenSession(): exception 0x40b00010, outer cleaning up...;
                      Locals/Code/MC/RTM/SMGNT/smgnt.c:895
Trustonic TEE: 104|MSH mshHandle_OPEN_SESSION(): can't start 00000000-1111-1111-1111-
                      111100000000;Locals/Code/MC/RTM/MSH/msh.c:73
Trustonic TEE: 104|MSH   Throw exception 0x40b00010 at;Locals/Code/MC/RTM/MSH/msh.c:75
```

These logs are printed when the System TA signing key used to sign the TA is not the one expected by the TEE. Note that this error is also usually visible thanks to the error code returned by the Normal World Client APIs: `MC_DRV_ERR_WRONG_PUBLIC_KEY` (value `0x00000019`, from MobiCoreDriverApi.h).

TEE Workspace out of memory

```
Trustonic TEE: 103|IPC   no free page available in pool 1;
                         Locals/Code/MC/RTM/MM/pagePool.c:428
Trustonic TEE: 103|IPC   Throw exception 0x40c00019 at;
                         Locals/Code/MC/RTM/MM/pagePool.c:428
```

Above is typical error printed by TEE Debug inmage in case of insufficient memory available in TEE Workspace. Around this error it should be easy to double check previous open/closeSession and review overall system memory usage.Trustonic Kinibi tool `Tee-Ps`, detailed later in this documentation could also be used.

TA trying to send command to a missing SWd driver

```
Trustonic TEE: 103|IPC handle_MSG_RQ_EX(): driver with ID -1530525925 not found;
                                           Locals/Code/MC/RTM/IPCH/ipch.c:958
Trustonic TEE: 103|IPC handle_MSG_RQ_EX(): returning E_TLAPI_DRV_NO_SUCH_DRIVER to 0xa03;
                                           Locals/Code/MC/RTM/IPCH/ipch.c:1059
```

In case a TA is trying to send a command to a SWd driver, if this SWd driver is not loaded (invalid SWd driver ID, service not loaded by a NWd Client…), then Kinibi will output above error message. For sure relevant error code will also be returned to the TA calling the drApiCallDriverEx() API.

Standard FIQ management logss

```
Trustonic TEE: mtk|MTK: starting intr      104
Trustonic TEE: mtk|MTK: intr 104 will be handled by thread 0x00000906
Trustonic TEE: mtk|MTK: start secure interrupt 104 to primary core #0
…
Trustonic TEE: mtk|MTK: stopping intr      104
Trustonic TEE: mtk|MTK: stop interrupt 104 (and make it non-secure)
Trustonic TEE: mtk|MTK: clear handler for intr 104
```

Logs printed by Kinibi TEE Kernel in case of SWd Interrupts (FIQ) management.

It is hard to list all potential error messages possible in Kinibi. However, the Kinibi TEE Debug image should output human readable error message for any possible case. If not enough, it will also precisely point into Kinibi source code the root of the error raised, allowing Trustonic Engineers to easily provide needed feedback.

4.2.3 TEE error codes returned to the TEE Linux driver

Even in release mode the Kinibi TEE Linux driver will output any errors received from the Secure World before re-forwarding it to the User Space libraries, usually under the format of either a **Legacy Trustonic Client API** or **GP Client API** error code.

Below is extract TEE Linux drivers logs from `dmesg`:

```
[ 22.698613]  [0:tee-mobicore-da: 2607] Trustonic TEE: mobicore_start:
[ 22.698613]  [0:tee-mobicore-da: 2607] Trustonic TEE: mobicore_start:
[22.698613]  [0:tee-mobicore-da: 2607]     product_id         = t-base-Android-400A-V016-20171128_104026_41079_71830
[22.698613]  [0:tee-mobicore-da: 2607]     version_mci        = 0x00010006
[22.698613]  [0:tee-mobicore-da: 2607]     version_so         = 0x00020002
[22.698613]  [0:tee-mobicore-da: 2607]     version_mclf       = 0x00020005
[22.698613]  [0:tee-mobicore-da: 2607]     version_container = 0x00020001
[22.698613]  [0:tee-mobicore-da: 2607]     version_mc_config = 0x00000003
[22.698613]  [0:tee-mobicore-da: 2607]     version_tl_api     = 0x00010014
[22.698613]  [0:tee-mobicore-da: 2607]     version_dr_api     = 0x00010004
[22.698613]  [0:tee-mobicore-da: 2607]     version_nwd        = 0x00060003
[22.698613]  [0:tee-mobicore-da: 2607]
...
[22.699096]  [0:tee-mobicore-da: 2607] Trustonic TEE: admin_open: daemon connection open, TGID 2607
[22.699096]  [0:tee-mobicore-da: 2607] Trustonic TEE: admin_open: daemon connection open, TGID 2607
[22.788571]  [1:    McDaemon.SWd: 2788] Trustonic TEE: admin_ioctl: daemon PID changed to 2786
[66.576225]  [3: mcDriverDaemon: 2548] Trustonic TEE: ERROR -1 mcp_cmd: open session: res 8
[85.750487]  [1:    tee_fastcall:  986] Trustonic TEE: mc_exec_core_switch: CoreSwap ok 1 -> 0
[85.750487]  [1:    tee_fastcall:  986] Trustonic TEE: mc_exec_core_switch: CoreSwap ok 1 -> 0
```

TEE Linux driver is first always dumping its version at TEE Linux driver probe time. Then, depending of the integration, more or less system information at runtime with specific error case highlighted in bold here.

Even if most of the time, the error code at this level is not so helpful (it's finally just a remapping), a complete definition of all the possible values returned by Kinibi TEE to the Normal World can be found in header file `\AndroidIntegration\Src\gud\MobiCoreDriver\mci\mcimcp.h`.

```
enum mcp_result {
       /** Operatin succeed */
       MC_MCP_RET_OK                                  =  0,
       /** The session ID is invalid */
       MC_MCP_RET_ERR_INVALID_SESSION                 =  1,
       /** The UUID of the Trustlet is unknown */
       MC_MCP_RET_ERR_UNKNOWN_UUID                    =  2,
       /** The ID of the driver is unknown */
       MC_MCP_RET_ERR_UNKNOWN_DRIVER_ID               =  3,
       /** No more session are allowed */
       MC_MCP_RET_ERR_NO_MORE_SESSIONS                =  4,
       /** The container is invalid */
       MC_MCP_RET_ERR_CONTAINER_INVALID               =  5,
       /** The Trustlet is invalid */
       MC_MCP_RET_ERR_TRUSTLET_INVALID                =  6,
       /** The memory block has already been mapped before */
       MC_MCP_RET_ERR_ALREADY_MAPPED                  =  7,
       /** Alignment or length error in the command parameters */
       MC_MCP_RET_ERR_INVALID_PARAM                   =  8,
       /** No space left in the virtual address space of the session */
       MC_MCP_RET_ERR_OUT_OF_RESOURCES                =  9,
       /** WSM type unknown or broken WSM */
       MC_MCP_RET_ERR_INVALID_WSM                     = 10,
…
```

For sure, any crash in TEE Linux driver would be considered as a bug and would need Trustonic to provide required fixes.

4.3 ERRORS FROM SWD DRIVERS

A Kinibi SWd Driver is relatively similar to a TA and in case of errors during the development, Kinibi is offering usual debug mechanisms like logging APIs or specific build options (`TRUSTLET_FLAGS/TA_FLAGS := 4`) to allow a SWd driver to print feedback or dump status in case of unexpected shutdown (SWd driver crash). For deeper information, a complete section is provided by Trustonic in *Kinibi_Driver_Developers_Guide.pdf*.

SWd Driver are running at SEL0 level and must go through the **Kinibi SWd Driver API** to access hardware or communicate with other system components.
Extract from file `\t-base-dev-kit\t-sdk\DrSdk\Out\Public\MobiCore\inc\DrApi\DrApiError.h` for error codes returned by **Kinibi SWd driver API**:

```
#define DRAPI_OK                        0x0      /**< Returns on success. */
#define E_DRAPI_KERNEL_ERROR            0xF01    /**< Kernel returned error. */
#define E_DRAPI_INVALID_PARAMETER       0xF02    /**< Invalid parameter. */
#define E_DRAPI_NOT_PERMITTED           0xF03    /**< Permission error */
#define E_DRAPI_IPC_ERROR               0xF04    /**< Error in IPC. */
#define E_DRAPI_TASK_NOT_ACCEPTABLE     0xF05    /**< Task not acceptable for operation. */
#define E_DRAPI_CANNOT_MAP              0xF06    /**< Cannot create mapping. */
…
```

However, SWd driver are supposed to manage specific hardware components and access very low-level platform registers. By consequence it is possible for a badly SWd Driver to crash or deadlock the entire system.
Normally Exceptions generated by a SWd Driver should be catch by its internal Exception Handling Thread. If not correctly handled the Exception is then catch by Kinibi RTM who will consider the system in an unrecoverable state and trigger a controlled Kinibi Halt.
In case of system deadlock detected (no answer to a command after too long period), the TEE Halt will be requested directly by the Linux TEE driver.

In both case the next sections should help the system integrators to dig more on origin of the issue.

5 DEBUGGING OF UNEXPECTED SWD EXCEPTIONS

Still in the `dmesg` log, in case of fatal TA crash, system deadlock or fatal SWd abort, the TEE will output different crash dump to help the system integrator to identify root cause of the halt. This section explains how to read the Kinibi TEE RTM/Micro Kernel crash dumps and try to explain what could happen there.

5.1 TEE TASKS EXCEPTIONS

When a task is closing in Kinibi system it is closed with one of the following exception code:

```
#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
#define TRAP_BREAKPOINT           5   // breakpoint
#define TRAP_ARITHMETIC           6   // arithmetic exception
#define TRAP_INSTR_FETCH          7   // instruction fetch failure
#define TRAP_INTERRUPT           16   // interrupt
#define TRAP_TIMEFAULT           17   // timefault exception
#define TRAP_TASK_TERMINATE      18   // child task termination
#define TRAP_TASK_ACTIVATE       19   // task activation
#define TRAP_TASK_START          20   // task start
```

The value to highlight is the `TRAP_TASK_TERMINATE`. It is the normal/usual value for the termination of a task, so it will be seen a lot of time.

5.1.1 User task exception

Using the TEE Debug image, in addition of many system logs, will provide similar output than having your TA built with the `TRUSTLET_FLAGS := 4` set. It means that all the Exceptions catch by the Kinibi RTM or kernel, normal termination, abort… will be displayed in the `dmesg`.

A typical Exception in the Kinibi looks like this.

- The TEE kernel prints the exception details and calls the registered exception handler. This can either be an internal thread of the task or the parent task's exception handler thread (typical expected case for correctly coded SWd driver, they should have an Exception thread handling the abort).
- If no exception handler is set up, the Kinibi RTM will get the exception and terminate the Trusted Application. As highlighted before, for unhandled driver exception, the RTM will shut down the system.

```
[188.134599] Trustonic TEE: 101(3)|FAULT in 701, thread 0x10007, UUID=cf27d291-427b-535c-a33d-
80c48bf3753b
[188.144215] Trustonic TEE: 101(3)|EXCH: trapType=0x2 (TRAP_SEGMENTATION), trapData=0x50001c,
cpsr=0x60000030 [USER,TCZ]
[188.155088] Trustonic TEE: 101(3)|EXCH: DataFault: DFSR=0x206 [TranslationL2,read], ADFSR=0,
DFAR=0x50001c [valid]
[188.165482] Trustonic TEE: 101(3)| r0=0x00000000,  r1=0x00000000,  r2=0x00000033,  r3=0x00000000,
[188.174417] Trustonic TEE: 101(3)| r4=0x00500018,  r5=0x00001777,  r6=0x001de1a0,  r7=0x001d2488,
[188.183328] Trustonic TEE: 101(3)| r8=0x00300010,  r9=0x00300008, r10=0x00500008, r11=0x001c87f9,
[188.192253] Trustonic TEE: 101(3)|r12=0x00500000,  sp=0x001d2300,  lr=0x00300010,  pc=0x001ca51c
[188.201079] Trustonic TEE: 101(3)|EXCH: ASLR offsets: task=0x001c7000, mcLib=0x07f1b000
```

So normally the exceptions should not be propagated outside of this dump (case of TA or SWd driver correctly handling their crash). In case of uncontrolled exception, logs detailed in next sections would appears… Leading mostly to a fatal halt of the complete SWd.

- The value for field `FAULT in` is providing the Process and Thread ID
  Warning: Process and Thread display have opposite order, Process is 4 digits based [2 Process iD][2 Thread iD], where Thread is 6 digits based [2 Thread iD][4 Process iD]
  This information will allow you to confirm which TA is exactly crashing (doing PiD/UUID mapping).
- The values `TrapType` and `TrapData` are the kernel's interpretation of the exception.
- `TrapType` is basically an abstraction from the ARM code exceptions.
- The `TrapData` value depends on the cause value in a way, that the kernel is trying to add meta-information there that is also passed to the exception handler. Unfortunately, this meta information is in some cases not enough to fully debug the exception. That is why the kernel prints more data to the debug output.
- The registers `r0-r12`, `sp`, `lr`, `pc`, `cpsr` hold the Generic Purpose registers snapshot when the crash occurred.
- `pc` points to the instruction of the crash, the TEE kernel does automatic adaption according to the ARM specs.

- The data fault registers `DFSR`, `ADFSR`, `DFAR` and instruction fault registers `IFSR`, `AIFSR`, `IFAR` do not always contain relevant values, it depends on the type of exception (`TrapType`), please refer to the ARM manual for details.
- Since Kinibi-410 and support of ASLR, values in registers are now only offset. Fields `ASLR offsets` should allow to find correspondence.

The expected way to investigate this type of issue is at the end like any system crash on ARM platforms. Looking at the exception type (seg fault, undef instr…) then disassemble TA binary and reviewing TAs instruction pointed by `ip`.
Usually, compiling a TA or a SWd driver with Kinibi sdk/ddk will automatically generate you the relevant symbols files `.lst2` extracted from unsigned TA binary (`.axf`). They can always be easily re-generated based on compiler toolchain:

```
# case of GCC toolchain
gcc-arm-none-eabi-4_8-2014q2/bin/arm-none-eabi-objdump -D
                        Out/Bin/GNU/Debug/taCryptoCatalog.axf >>
                        Out/Bin/Debug/taCryptoCatalog.lst2
# case of RVCT toolchain
ArmRvct/5.04b82/Out/bin/fromelf --text -c -d -e -t -z
                --datasymbols Out/Bin/Debug/taCryptoCatalog.axf
                --output Out/Bin/Debug/taCryptoCatalog.lst2
```

Addresses provided by the instruction pointer or link register (`lr=0x00007787` or `ip=0x00007f20`) can then be searched in the symbol table of the `.lst2` and provide you exact place of crash for the `ip` and function where we were supposed to return in `lr`.

Note that in Kinibi-400 and previous TEE versions, TA/SWd drivers are all loaded at fixed virtual addresses, starting from the address `0x0000`**`1xxx.`** More details on TEE pre-defined memory ranges are available in next section of this chapter.
Warning: It's not the case in Kinibi-410 which is introducing ASLR for the TAs and the Kinibi McLib. In that case, crash dump is providing the input to

5.1.2 Kinibi RTM exceptions

Exception in Kinibi RTM threads are only presented for informative reason as this is absolutely not supposed to happen (RTM can only be accessed through well-defined API that should always return correctly with success/error code). The RTM exception handler is called and it will terminate the RTM as default action.

```
MTK: EXCEPTION in thread 0x40001, cpsr=0x30 [USER,T]
MTK: cause=0x2 (TRAP_SEGMENTATION), meta=0x7
MTK: DFSR=0x7 [TranslationSectionPage,read,dom=0], ADFSR=0, DFAR=0x7
MTK: IFSR=0x5 [TranslationSection], AIFSR=0, IFAR=0x48000cc0
MTK: r0=0x00203470, r1=0x00000478, r2=0x0000045c, r3=0x00000002
MTK: r4=0x00200000, r5=0x00000003, r6=0xffffffff, r7=0x0001f5cc
MTK: r8=0x00000004, r9=0x00000001, r10=0x40b0000a, r11=0x00300000
MTK: r12=0x0001d79c, sp=0x0001f528, lr=0x0000693b, ip=0x0000694c
exchLoop():         EXCEPTION         in         RTM         thread=0x00040001,         type=0x00000002,
data=0x00000007;Locals/Code/MC/RTM/EXCH/exch.c:134
RTM Exception: ### MOBICORE HALT ###
```

The worst exception in RTM is an exception in the exception handler. For sure this is also not supposed to happen.
It's very likely that such things happen due to a bug in RTM, e.g. a missing parameter check. In the worst case a previous bug has corrupted the RTM memory anyway, so the state is unrecoverable now. The kernel will terminate the RTM and shut itself down - it has no other choice, as the init task (RTM) is never supposed to run into any unhandled exception.

```
MTK: EXCEPTION in thread 0x20001, cpsr=0x20000010 [USER,C]
MTK: no exception handler, terminating task 1
MTK: stopping intr 161
MTK: clear intr handler 161
MTK: Sigma0 terminated by thread 10001
MobiCore/MTK: ### SYSTEM HALT, code=2
MTK: HALT, ip=1f03018, code=2
```

5.1.3 Kinibi TEE Kernel Panic

Like the exception in Kinibi RTM, this type of error should never happen and would highlight mainly a TEE bug.

```
MTK: ### KERNEL PANIC, cpsr=0x60000093 [SVC,ICZ]
MTK: cause=0x2 (TRAP_SEGMENTATION), meta=0x100000
MTK: DFSR=0x406 [AsyncExtAbort,read], ADFSR=0, DFAR=0x100000 [invalid]
MTK: IFSR=0x5 [TranslationSection], AIFSR=0, IFAR=0x2028c40
MTK: r0=0x00000000, r1=0x00000000, r2=0x00000020, r3=0x00000000
MTK: r4=0x00001215, r5=0x00100fff, r6=0x00006bb8, r7=0x00000009
MTK: r8=0x00000000, r9=0x00000000, r10=0x00006bac, r11=0x00000000
MTK: r12=0x000fd625, sp=0x00006b78, lr=0x000fd67d, ip=0x01f00008
MobiCore/MTK: ### SYSTEM HALT, code=4001
MTK: HALT, ip=1f055b4, code=4001
```

Final actions are similar also, entering in TEE system halt, all SMCs will be rejected.

5.2 KINIBI TEE HALT SYMPTOMS

This SWd dump is printed in NWd logcat if the TEE is down:

```
[ 8109.496321] Trustonic TEE: MTK: SIQ, but system halted
[ 8118.408502] Trustonic TEE: wait_mcp_notification() ### ERROR: No answer after 10s

[ 8118.415441] Trustonic TEE: <t-base halted. Status dump:
[ 8118.419963] Trustonic TEE:   flags               = 0x80000401
[ 8118.426080] Trustonic TEE:   haltCode            = 0x00000002
[ 8118.431642] Trustonic TEE:   haltIp              = 0x07f03496

[ 8118.437598] Trustonic TEE:   faultRec.cnt        = 0x00000002
[ 8118.443079] Trustonic TEE:   faultRec.cause      = 0x00000002
[ 8118.448487] Trustonic TEE:   faultRec.meta       = 0x00000000
[ 8118.454146] Trustonic TEE:   faultRec.thread     = 0x00010003
[ 8118.459869] Trustonic TEE:   faultRec.ip         = 0x00004f2e
[ 8118.465635] Trustonic TEE:   faultRec.sp         = 0x00009ce8
[ 8118.471317] Trustonic TEE:   faultRec.arch.dfsr  = 0x00000807
[ 8118.477075] Trustonic TEE:   faultRec.arch.adfsr = 0x00000000
[ 8118.482746] Trustonic TEE:   faultRec.arch.dfar  = 0x00000000
[ 8118.488525] Trustonic TEE:   faultRec.arch.ifsr  = 0x00000005
[ 8118.494185] Trustonic TEE:   faultRec.arch.aifsr = 0x00000000
[ 8118.499952] Trustonic TEE:   faultRec.arch.ifar  = 0xbe001430
[ 8118.499953] Trustonic TEE:   faultRec.offset.task= 0x00000000
[ 8118.499955] Trustonic TEE:   faultRec.offset.mclib=0x07e38000

[ 8118.505624] Trustonic TEE:   mcData.flags        = 0x00000001
[ 8118.511498] Trustonic TEE:   mcExcep.partner     = 0xffffffff
[ 8118.517067] Trustonic TEE:   mcExcep.peer        = 0x00010003
[ 8118.522833] Trustonic TEE:   mcExcep.cause       = 0x00000002
[ 8118.528515] Trustonic TEE:   mcExcep.meta        = 0x00000000
[ 8118.534312] Trustonic TEE:   mcExcep.uuid        = 0x07050500000000000000000000000020
```

This dump is organized in 3 different areas:

- The *TEE Kernel global status*, giving current global kernel situation (running, idle, crashed…);
- The *TEE Kernel Exception status*, tracking data of last exception raised in TEE Kernel;
- The *TEE RTM Exception* status, tracking data of last exception raised in TEE RTM.

The goal is to provide the status of the 2 main Kinibi system components at the time of the exception, without leaking any sensitive information. Data in this status can or cannot overlap depending of exception context (RTM itself generating a TEE halt in case of unrecoverable scenario or TEE Kernel crashing)

Next sections will provide more details interest/goal of these fields.

Note that the TEE Linux driver has also addition mechanism to recover some of the MTK state, relying on "Linux debugfs support" to allow NWd task to obtain the crash info. However, understanding the output requires knowledge of the SWd kernel internals, such information may not be available to 3rd parties or even SIPs/OEMs.

5.2.1 Kinibi Kernel global status at Exception time

```
[ 8118.419963] Trustonic TEE:   flags              = 0x80000401
```

```
Overall Kinibi TEE Kernel status flag:
SYS_STATE_RUNNING 0x80000201 : system running (a task is schedulable)
SYS_STATE_IDLE    0x80000301 : system idle    (no task is schedulable)
SYS_STATE_HALT    0x80000401 : system halted  (no task will run)
```

```
[ 8118.426080] Trustonic TEE:   haltCode           = 0x00000002
```

```
Kinibi TEE Kernel halt code.
Most common values will be:
- SYS_OK,                      0x0000 /* Not halted */
- SYS_HALT_SIGMA0_TERMINATE,   0x0002 /* Controlled shutdown of RTM. */
- SYS_HALT_ARCH_KERNEL_PANIC, 0x4001 /* Kernel halt */
- SYS_HALT_ARCH_KERNEL_PANIC, 0x4010 /* Fastcall hook halt – newly added in Kinibi-410 */
```

```
[ 8118.431642] Trustonic TEE:   haltIp             = 0x07f03496
```

```
IP of the function in the Kinibi Kernel when halt occurred.
(Most of time, RTM is not the origin of the issue and so will point on the code in charge of this TEE
status dump)
```

Any analysis of a SWd halt has to start with the 3 status field above. The value of field `haltCode` will drive the next steps:

1) Value is 0x0 means that TEE has not crashed.
   If NWd is taking decision to dump TEE status, it's mostly because TEE has not responded in time and it could highlight a case of SWd deadlock (like a badly written SWd Driver, never re-entering in API `drApiIpcCallToIPCH()`, a crash or a lock in a Fastcall Hook blocking TEE kernel to update dump status)
   In this case all following fields could give few information on overall context… But will mostly not point the root cause of the deadlock.
2) Value 0x02 is the error code raised when the Kinibi RTM is taking decision to halt the SWd.
   It could be linked to a bug (a TA calling a TEE API which could crash).
   It could be a side effect of a SWd Driver itself crashing and badly implementing its exception handling thread.

In all these situations, analysis of complete dump should allow to gather more information on origin of the issue (task ID, TA UUID, Instruction Pointer…). Next step most of the time is to jump directly to Kinibi RTM Exception dump before digging into Kinibi Kernel Exception status.

3) Value 0x4001 means that TEE Kernel directly crashed. It should not happen and mostly highlight a deep system issue (TEE Kernel bug, Fastcall hook issue…).
   Again, analysis of complete dump should allow to gather more information on origin of the issue (task ID, TA UUID, Instruction Pointer…)
4) Value 0x4010 has been specifically added in Kinibi-410, to easier track specific crashes in context of a Fastcall hook. In this situation, only the `haltIp` flag should be considered.

5.2.2 Kinibi Kernel Exception status

```
[ 8118.437598] Trustonic TEE:   faultRec.cnt        = 0x00000002
```

This is the number of faults recorded in Kinibi since boot. Standard values:

- 0x1, is the normal/expected value (due to Kinibi CyptoDriver performing a self-test during initialization).
- 0x2 or more, means we have an additional and mostly unexpected fault.

```
[ 8118.443079] Trustonic TEE:   faultRec.cause      = 0x00000002
```

This is the fault cause recorded:

```
#define TRAP_UNKNOWN         ( 0)  /**< 0x0, unknown exception. */
#define TRAP_SYSCALL         ( 1)  /**< 0x1, invalid syscall number. */
#define TRAP_SEGMENTATION    ( 2)  /**< 0x2, illegal memory access. */
#define TRAP_ALIGNMENT       ( 3)  /**< 0x3, misaligned memory access. */
#define TRAP_UNDEF_INSTR     ( 4)  /**< 0x4, undefined instruction. */
#define TRAP_BREAKPOINT      ( 5)  /**< 0x5, breakpoint. */
#define TRAP_ARITHMETIC      ( 6)  /**< 0x6, arithmetic exception. */
#define TRAP_INSTR_FETCH     ( 7)  /**< 0x7, instruction fetch failure. */
#define TRAP_INTERRUPT       (16)  /**< 0x10, interrupt. */
#define TRAP_TIMEFAULT       (17)  /**< 0x11, timefault exception. */
#define TRAP_TASK_TERMINATE  (18)  /**< 0x12, child task termination. */
#define TRAP_TASK_ACTIVATE   (19)  /**< 0x13, task activation. */
#define TRAP_TASK_START      (20)  /**< 0x14, task start. */
```

```
[ 8118.448487] Trustonic TEE:   faultRec.meta       = 0x00000000
```

This is the address/value where something bad happened.
So, depending of context (faultRec.cause):

case "TRAP_UNDEF_INSTR", gives IP of Undef exception
case "TRAP_INSTR_FETCH", gives address of instr fetch
case "TRAP_SEGMENTATION", gives address of segfault
case "TRAP_ALIGNMENT", gives address of alignment fault
case "TRAP_SYSCALL", gives syscall ID failed

```
[ 8118.454146] Trustonic TEE:   faultRec.thread     = 0x00010003
```

The Kinibi Process/Thread that caused the fault. Here, thread 1 of task 3 (this is DrSecureFS).
Cross-reference the information displayed by:

- In the log header of any SWd traces ("*Trustonic TEE: 104|MSH searching service in TEE image"*, 104 meaning thread 4 of task 1)
- TEE Debug image during TA or SWd driver openSession()

```
[ 8118.459869] Trustonic TEE:   faultRec.ip         = 0x00004f2e
```

The Instruction pointer of the task that was running when the exception happened.
Cross-reference the TA/SWd driver .lst2 file for more information.

```
[ 8118.465635] Trustonic TEE:   faultRec.sp         = 0x00009ce8
```

The stack pointer at the time of the fault

```
[ 8118.471317] Trustonic TEE:   faultRec.arch.dfsr  = 0x00000807
[ 8118.477075] Trustonic TEE:   faultRec.arch.adfsr = 0x00000000
[ 8118.482746] Trustonic TEE:   faultRec.arch.dfar  = 0x00000000
[ 8118.488525] Trustonic TEE:   faultRec.arch.ifsr  = 0x00000005
[ 8118.494185] Trustonic TEE:   faultRec.arch.aifsr = 0x00000000
[ 8118.499952] Trustonic TEE:   faultRec.arch.ifar  = 0xbe001430
[ 8118.499953] Trustonic TEE:   faultRec.offset.task= 0x00000000
[ 8118.499955] Trustonic TEE:   faultRec.offset.mclib=0x07e38000
```

Dump of usual fault status registers.

In this example, a segfault.

This part of the SWd halt dump is focusing on TEE Kernel exception data. The first field here to look at is the fault counter, faultRec.cnt. A value equal or superior to 0x2 is confirming an un-expected fault (either inside the TEE itself or potentially a side effect from a SWd driver).

5.2.3 Kinibi RTM Exception status

```
[ 8118.505624] Trustonic TEE:   mcData.flags        = 0x00000001
[ 8118.511498] Trustonic TEE:   mcExcep.partner     = 0xffffffff
[ 8118.517067] Trustonic TEE:   mcExcep.peer        = 0x00010003
[ 8118.522833] Trustonic TEE:   mcExcep.cause       = 0x00000002
[ 8118.528515] Trustonic TEE:   mcExcep.meta        = 0x00000000
```

Dump of RTM Exception status, meaning:

- Exception raised from task/thread mcExcep.peer.
- Exception caused by mcExcep.cause, with additional info in mcExcep.meta.

Where values are those for fields faultRec.cause and faultRec.meta.

```
[ 8118.534312] Trustonic TEE:   mcExcep.uuid        = 0x07050500000000000000000000000020
```

TA or SWd driver UUID matching thread/TaskID in mcExcep.peer.

In this section of the dump the first field to focus is the mcExcep.cause:

- Seeing the value 0x12, TRAP_TASK_TERMINATE, means that last RTM exception registered is the closing of a task. It is an expected value and not highlighting an abnormal behavior (just showing last TA closed… meaning mostly this part of the dump is not more interesting).
- Any other values will highlight an exception catch by the RTM, who then have forced the TEE Kernel to enter in halt mode.
  In this situation:
  - Cross reference with Kinibi Kernel global status field haltCode, which is usually set to 0x2;
  - Kinibi Kernel Exception handler should point to the exact fault instruction (faultRec.ip).

5.2.4 Addresses ranges in Kinibi without ASLR

This section is informative and subject to change depending of TEE versions. Values are provided mainly for informative reason and it is highly recommended to double check with Trustonic support team.

As said earlier, until Kinibi-400, software components are loaded at fixed virtual addresses. Even if debug symbols are not publicly available for all components (Kinibi internal TEE binary), having a view on these values/ranges could help system integrators to localize origin of the issue:

Binary considered	Loaded range
TA/SWd driver symbols	0x0000 **1xxx**…
MCLib symbol (APIs implementation).	0x0**7dx xxxx**
RTM symbols (system services)	0x0000 **1xxx**…
MTK symbols (TEE Micro Kernel)	0x0**7Fx xxxx**

These assumptions are false on Kinibi-410 where TA/SWd driver and MCLib are loaded at random addresses, changing at each TA loading. In that case, the TA crash dump will provide needed relocation information.

5.2.5 Debugging on a Kinibi with ASLR (Kinibi-410)

With ASLR, code section moves in a 2MB range and McLib code moves in a 2MB range. This means the PC reported by the crash dump can be modified by the ASLR offset. A complete guide on TA debugging in the face of ASLR is provided by Trustonic in *Kinibi_Developers_Guide.pdf*. As stated there, during TA development phase, you can use the TA_DEBUGGABLE flag to get ASLR offsets on TA open-session and on TA crash. During system integration phase, if you do not use TA_DEBUGGABLE, you can use the Kinibi Kernel exception status to get the ASLR offset for TA code and McLib code.

```
[ 8118.459869] Trustonic TEE:   faultRec.ip          = 0x001ca51c
```

The Instruction pointer of the task that was running when the exception happened.
Cross-reference the TA/SWd driver.lst2 file for more information, but using the ASLR offsets:

```
[ 8118.499953] Trustonic TEE:   faultRec.offset.task = 0x001c7000
[ 8118.499955] Trustonic TEE:   faultRec.offset.mclib= 0x07e38000
```

From this trace we can see:

- The PC is at 0x001ca51c
- The TA code was relocated to 0x001c7000.
- The McLib code was relocated to address 0x07e38000.

From this information we can deduce:

- PC was in TA code, not in McLib (PC smaller than mcLib offset)
- Original PC is 351c = 0x001ca51c - 0x001c7000 = faultRec.ip - faultRec.offset.task
- If PC or LR are in McLib code, then McLib offset needs to be subtracted.

5.3 LOCKING TEE ON ONE CORE (KINIBI-410)

Kinibi-410a introduces the automatic core switch, where the SMC instruction could be issued by any CPU, only depending on Linux scheduling strategy. This can pose challenges to debug applications and the system. The following command-line options may help to pin the TEE on one core, thereby disabling the automatic core switch.

- Determine PID of tee_scheduler:

```
$ adb shell "grep tee_scheduler /proc/*/sched"
 /proc/3334/sched:tee_scheduler (3334, #threads: 1)
```

On different Linux this can give:

```
 /proc/2742/status:Name:    tee_scheduler
```

PID is 3334 or 2742.

- Then set affinity of this PID to core 0 only:

```
$ adb shell "taskset -p 1 2742"
 pid 2742's current affinity mask: ff
 pid 2742's new affinity mask: 1
```

Bring the TEE back to run on any core:

```
$ adb shell "taskset -p ff 2742"
 pid 2742's current affinity mask: 1
 pid 2742's new affinity mask: ff
```

Of course, you can also make the TEE run on other cores, by setting the mask to 2 or 4, 8, 10, 20, 40, 80.

It is also possible to allow several cores by using the mask feature.

6 SWD STATE WITH KINIBI TOOL TEEPS

The Kinibi product package contains the `tee-ps` tool in `/t-base-dev-kit/Tools/TlcTeePs` with source code and Android ARM binaries. The purpose of this tool is to provide to Kinibi developers/integrators feedback on overall SWd status: Sessions running and info, memory consumptions per service, last messages exchanged with TEE system (no data, only message type). Usage could either be system debug to understand overall SWd status or either overall system benchmarking.

On startup, the mcDriverDaemon (Daemon) opens the TEE DebugSession in Kinibi. Using this session, the Daemon shares a buffer with the TEE. The buffer is mapped in Kinibi RTM and in MTK.
During the run time:

- the RTM updates this buffer with information about sessions including session ID, UUID, memory usage and last IPCH call.
- The MTK updates the buffer with information about threads including the threadid, last syscall, and last modified time.

Also, in case of TEE crash, the file "`/data/local/tmp/debugsession`" will automatically be created. Then, no need to call `tee-ps` to send a signal to mcDriverDaemon. The file can be directly parsed later with `tee-ps -I`

Before entering more in tool details and information printed it is important to consider the following concepts:

Microkernel – Threads - IPC

In Kinibi, the *MTK* is a microkernel that isolates tasks from each other and implements multithreading for drivers and TAs. A task can have one or several threads. Each thread has a threadID. Threads can do a limited number of system calls to MTK. Using the IPC system call, a thread can send a message to another thread by specifying the threadID of the target thread. Inter-process communication (IPC) is synchronous, it means that either the sender or the receiver must block and wait for the other one to finish the IPC system call. In general, we speak of an "IPC send" when thread A sends a message to thread B. The thread B needs to do an "IPC receive" to receive the message. Often the two are combined in client-server communication: The client will do "IPC send" to server and immediately afterwards an "IPC receive" from server – this is called *IPC call* from client to server. The server will usually do "IPC receive" from anyone, wait for a request, handle it, and then do an "IPC send" to the client, immediately followed by a new open "IPC receive".

These notions are important because in case of deadlock, often one thread is waiting for another and the DebugSession can show this situation.

TEE DebugSession – kernel updates

The kernel area of the DebugSession buffer contains a header and an array with information about all threads. The header shows the current and last thread – this is useful to find out which thread, task, TA or Driver was scheduled last. It could be the one that provoked a crash.

For the microkernel, all events are generated by threads that perform system calls. That is why the DebugSession gives detailed information about each thread. In effect, on each system call, the kernel will update the debug information of the affected threads.

Fastcalls

In addition to threads, the MTK also handles fastcalls from the Normal world. The DebugSession buffer has a history buffer for fastcalls, however, this buffer is not activated today in Release version of the product.

RTM – Run time manager

The RTM manages the high-level concepts and can be seen as "the TEE". The RTM is the first task started by the kernel and manages the system resources. It will create tasks for TAs and Drivers, manages the memory allocation and the communication with the NWd. All the TAs and Drivers have to communicate via RTM and cannot communicate directly with each other. The RTM has itself several threads, called EXCH, SIQH, IPCH, MSH, LOADH that each handle specific events. The taskid of RTM is 1, the threadID of Exch 101, SIQH 102, IPCH 103, MSH 104, LOADH 105 in the dmesg traces. In the MTK, the threadIDs are 00010001, 00020001, 00030001, 00040001, 00050001 – the taskid is the lower 16 bit of the threadID.

Booting the TEE

The Exch is the initial thread of RTM and will initialize RTM and start RTM threads. It then starts the embedded services: the crypto driver 201, 202 and the Secure Filesystem Driver 301. Afterwards all threads of the RTM become idle. The MTK will return to the NWd. Later during the boot Mobiload will create an MCI connection with RTM, load drivers and close the MCI connection. Much later the Linux kernel will again create the MCI connection with RTM. Eventually the McDriverDaemon will start the TEE services: the Filesystem Daemon will open a session to the Secure Filesystem Proxy TA and the DebugSession thread will open a session to RTM.

TEE DebugSession – RTM Exception handling

When a TA exits or crashes, the RTM exception handler *EXCH* receives an IPC message. The DebugSession contains a history buffer of the last 5 exceptions as seen by EXCH.

TEE DebugSession – RTM notification handler

When a Client Application in NWd sends a notification to a TA, or when an MCP control message is sent ready for the TEE, the RTM software IRQ handler *SIQH* receives an IPC message from the kernel. The DebugSession contains the last notification of SIQH, including the two fields, sessionID and payload.

TEE DebugSession – RTM session handler

When a Client Application in NWd sends a MCP control message, like MCP_OPEN_SESSION, the RTM master session handler *MSH* receives an IPC message from SIQH. The DebugSession contains the last MCP command, the last opened session, the last closed session and the last error reported by MSH.

TEE DebugSession – RTM IPC handler

When a Trusted Application or a Secure Driver call an API it mostly creates an IPC call to RTM inter process handler *IPCH*. The IPCH handles the notifications, driver calling, memory management and session management for the TAs and Drivers. The DebugSession contains the last caller of IPCH. Also, for each session in RTM, the DebugSession contains the last command and its first parameter that the TA or Driver has requested from IPCH.

TEE DebugSession – RTM session information

The RTM is managing the session in the TEE and the DebugSession buffer contains a solid information about each session. When a session is opened, the RTM enters an entry into the DebugSession with the identity information of this session, like UUID, SPID, RootID, sessionID, serviceType, driverID, isGP. The RTM will also enter the resources used by the session, such as threadID, secondaryThreadID, number of memory pages used, number of MMU tables used, accumulated size of the TA sections, like code, data, heap, world-shared memory mappings, physical memory mappings, client mappings. The RTM updates the resource information whenever the CA, TA or Driver make a memory-related request to IPCH. Also, as stated above, the RTM records the last IPCH command that a TA or Driver has send to RTM.

`Tee-ps` available options:

```
user@x86-pc:~/t-base-dev-kit/Tools/TlcTeePs$ ./Out/Bin/x86/gcc/teeps-x86 -h

usage: teeps-x86 [-hakfmorst]
Copyright (c) Trustonic Limited 2015-2017.
lukhan01@DE0015-20180312_182439_75400
-i file         Dump file to read
-a/A            ALL (product ID, RTM information, kernel information and FastCall buffer)
-f/F            FASTCALL (FastCall buffer)
-k/K            KERNEL (kernel information)
-m/M            dggFs last mcp commmands
-o/O            KERNEL ORDERED (kernel information with threads ordered by last modification)
-g/G            KERNEL GROUPED (kernel information with threads grouped by session)
-r/R            RTM (RTM information)
-s/S            dbgFs structs and sessions
-t/T            TOP (executes a 'Linux top' like action on the corresponding command, only first argument is taken in account)
                      -> eg : ./tee-ps -tr  (RTM info dynamically)
```

```
                    -> eg : ./tee-ps [-tfr | -frt] (only fastcall info dynamically)
-no option      Product ID and RTM information
```

`Tee-ps` can send a specific signal to the Kinibi `mcDriverDaemon` to tell the Daemon to dump the data into a file (by default). The format of the data is a C structure with many fields and arrays of structures. The `tee-ps` command-line tool can read the DebugSession data and print its content in a string format to the user.

In "top" mode, `tee-ps` regularly sends a signal to the Daemon and refreshes the screen output with the last file content as written by the Daemon. Note that running the `tee-ps` tool requires elevated privileges like root.

6.1 RTM SESSION INFORMATION

The RTM session information can be extremely usefully in overall system benchmarking, providing a lot of feedback on sessions currently running and memory usage of all of them. In case of post mortem debug it's also providing interesting status for each task on his last syscall. Below is a subset sample of logs you can get:

```
601: System TA
 SPID=0xfffffffb, UUID=08050500-0000-0000-0000-000000000000
 State=Active, flags=0x00000000, max instances=1
 Uses a total of 157 pages (628 KB), 4 tables:
  BlobRO=0/0, BlobRW=41/164, Client=0/0,
  WSM=256/1024, Phys=0/0, Heap=0/0, Misc=104/416
 IPCH info: ID=0, len pages=0, mr0=0x0, mr2=0x0
```

The global session info

- `601` – The session ID
- `System TA` – the type of the service, could also be `Driver`
- `SPID` – the SPID
- `UUID` – the UUID, as in the TA makefile and MobiConvert parameter
- `State` – Active

```
SESSION_STATE_INITIALIZING   = 1, /**< Session is going up */
SESSION_STATE_ACTIVE      = 2, /**< Session is up and running (open) */
SESSION_STATE_CLOSING     = 3, /**< Session closing has been started by MSH/CA */
SESSION_STATE_CLOSED      = 4, /**< Session closing has been finished by EXCH */
```

- `Flags` – the Flags from MCLF header

```
MC_SERVICE_HEADER_FLAGS_PERMANENT            1 Loaded service cannot be unloaded
MC_SERVICE_HEADER_FLAGS_NO_CONTROL_INTERFACE 2 Service has no WSM control interf.
MC_SERVICE_HEADER_FLAGS_DEBUGGABLE           4 Service can be debugged
MC_SERVICE_HEADER_FLAGS_EXTENDED_LAYOUT      8 New-layout TA or SWd driver
```

So overall translation for `flags=0x00000000`:

- not permanent (normal case for a TA),
- has TCI (normal case for a TA),
- not debuggable (normal for production device),
- not extended memory layout – Not recommended, new layout should be envisaged

- `Max instances` – as from the MCLF header: how many sessions can be open at the same time to this TA. Usually 1 for drivers, it can be more for TAs.

Session Memory Usage

- `Uses a total of 157 pages (628 KB), 4 tables` – Indication of the size of the TA. However, note that WSM, physical mappings and client mappings are counted here as well. The MMU tables consume 4KB from a separate table pool.
- `BlobRO` – Size of the code area of this TA.
- `BlobRW` – Size of the data and BSS area of this TA.
  - `41 / 164` – it means 41 pages corresponding to 164 KB.
- `Client` – For a driver, size of client mappings as created by drApiMapTaskBuffer().
- `WSM` – Size of the world-shared memory, i.e. the TCI or the memory shared from NWd with mcMap.
- `Phys` – For a driver, size of device mappings as created by drApiMapPhysicalBuffer().
- `Heap` – Size of the used heap, allocated by STACK macro and TEE_Malloc().

Session Messaging info

- `IPCH info` – information about last message send from this session to RTM IPCH.
- `ID=6` – Message ID, as seen in DrApi/DrApiIpcMsg.h message_t enum.

```
// Used for initializing state machines
MSG_NULL                        = 0,
MSG_RQ                          = 1,
// Client Request, blocks until MSG_RS is received
// Client -> Server
MSG_RS                          = 2,
// Driver Response, answer to MSG_RQ
```

```
// Server -> Client
MSG_RD                          = 3,
// Driver becomes ready
// Server -> IPCH
MSG_NOT                         = 4,
// Notification to NWd for a session, send-only message with no
// response
// client/server -> IPCH;
MSG_CLOSE_TRUSTLET              = 5,
// Close Trustlet, must be answered by MSG_CLOSE_TRUSTLET_ACK
// MSH -> IPCH, IPCH -> Server
MSG_CLOSE_TRUSTLET_ACK          = 6,
// Close Trustlet Ack, in response to MSG_CLOSE_TRUSTLET
// Server -> IPCH
…
MSG_CLOSE_DRIVER                = 9,
// Close Driver, must be answered with MSG_CLOSE_DRIVER_ACK
// MSH -> IPCH, IPCH -> Driver/Server
…
MSG_RQ_EX                       = 27,
// Client Request, blocks until MSG_RS or MSG_RQ_ERROR is received
// Client -> Server
…
MSG_WAIT_EVENT                  = 45,
// Tell the RTM that a session is waiting for an event
MSG_EVENT                       = 46,
// Send a notification event
```

- `len pages=0` – When the TA calls tlApi_callDriverEx(id, buffer, size), the size parameter is in len pages. Page length is only valid for MSG_RQ_EX.
- `mr0=0x40001` – Parameter to the MSG, it depends on the MSG.
- MSG_RQ and MSG_RQ_EX: Driver ID when the TA calls tlApiCallDriverEx(id, buffer, size).
- `mr2=0xffffadd5` – always blanked out in Release mode, it means "0xffff address".

Special notes:

- The UUID=08050500-0000-0000-0000-000000000000 is a pseudo session and represents the DebugSession itself. It does not consume resources.
- Embedded services have a BlobRO size of 0, because no memory needed to be allocated to start them.
- See the Driver developer guide on MSG_RD, MSG_RS, MSG_RQ / MSG_RQ_EX flow.

6.2 TEE KERNEL INFORMATION

6.2.1 TEE SMCs history

The TEE Kernel data is offering first an history on the last SMCs instructions received:

```
Dumping Kernel entry/exit time (oldest first)
   Entry time(us)   Exit time(us)   SWd(us)   Nwd(us)   Cycles   Freq(~KHz)   Entry   Exit   Cpu    Thread ID
0)    0                0              0         0         0         0          ?       ?     0        0
1)    0                0              0         0         0         0          ?       ?     0        0
…
97)   57522866         57522955       89        55        67513     758        nSIQ   YIELD    5    20001
98)   57522980         57523036       56        25        42314     755        nSIQ   YIELD    5    20001
99)   57523059         57523060        1        23          999     999       YIELD   YIELD    5    20001
```

- `Entry time(us)`, for the SWd time at SMC entry;
- `Exit time(us)`, for the SWd time at SMC exit;
- `SWd(us)`, time spent in SWd for this SMC command (computed from previous values);
- `Nwd(us)`, time spent in NWd since previous SMC command (computed from previous values);
- `Cycles`, number of cycles spent in SWd for this SMC command (extracted from ARM perf counters);
- `Freq(~KHz)`, informative freq, basd on previous time/cycles;
- `Entry`, SMC used to enter in SWd (`YIELD` == restart last, `nSIQ` == go through scheduler);
- `Exit`, Event motivating SWd exit (`YIELD` == cmd finished nothing else to do, `IRQ` == command interrupted by NWd);
- `CPU`; the CPU who have handled the SMC;
- `Thread ID`, the Thread ID finally scheduled during execution of this SMC.

The TEE SMC history can be helpful information in case of system hang or post mortem debug, giving deep information on last TEE activity (last entry in SWd, last SWd scheduling time, reason for exit…).

For example, in case of overloaded system due to IRQ storm, we would see many exit reasons tagged with `IRQ`, with SWd execution time reduced to very small values.

6.2.2 TEE crashes/aborts history

```
Dumping crashes (oldest first). Total = 0
      Thread ID          PC          SP    taskOff   mcLibOff
0)            0           0           0          0          0
1)            0           0           0          0          0
2)            0           0           0          0          0
3)            0           0           0          0          0
4)            0           0           0          0          0
5)            0           0           0          0          0
6)            0           0           0          0          0
7)            0           0           0          0          0
8)      0x30005      0x2322      0x7300          0  0x07e9d000
9)      0x10005      0x179a      0x4bd8          0  0x07e3a000
```

The TEE SMC history buffer is simply tracking the 10 lasts crashes in order to rebuild the chain in case of fatal exception.

6.2.3 TEE Kernel Threads data

The threads information is highly dependent on the microkernel. If these data could be useful to get feedback on execution time for each task, it's also providing deep TEE kernel status mainly needed in case of complex/post mortem debug, see for example:

```
        501 Thread Id=0x00010005 at PC=0x07d02552, SP=0x000040c0
        flags=0x00000104, ipcPartner=0x00000000, last syscall=SIGNAL_WAIT(20),
        last modified=119846168 ms, total user time=12 µs, total sys time=136 µs
```

- `501` – the session ID
- `Thread ID=0x00010005` – the actual threadID
- `PC=0xffffc0de` – the last used program counter in the TA code.
  It is blanked out in release mode, 0xffff "code".
- `SP=0xffffda7a` – the last used stack pointer in the TA execution.
  It is blanked out in release mode, 0xfff "data".

Scheduling attributes

These 3 flags are not really helpful and mainly targeting Trustonic in order to get feedback from TEE kernel in case of post mortem debug. Analysis require a deep TEE Kernel knowledge.

- `flags=0x00000104` – thread status for the microkernel
  flags is bitfield mixing schedule status and priority

```
#define THREAD_FLAG_PRIORITY_MASK   (0x000000ff)

#define THREAD_FLAG_SCHED_ENQUEUED  (1U << 8)  0x00000100
#define THREAD_FLAG_TIME_ENQUEUED   (1U << 9)  0x00000200
#define THREAD_FLAG_TIME_FAULT      (1U << 10) 0x00000400
#define THREAD_FLAG_SYS_RESTART     (1U << 16) 0x00010000
#define THREAD_FLAG_IPC_WAIT_TX     (1U << 17) 0x00020000
#define THREAD_FLAG_IPC_TX_DONE     (1U << 18) 0x00040000
#define THREAD_FLAG_IPC_RX_STARTED  (1U << 19) 0x00080000
#define THREAD_FLAG_IPC_WAIT_RX     (1U << 20) 0x00100000
#define THREAD_FLAG_IPC_RX_DONE     (1U << 21) 0x00200000
#define THREAD_FLAG_IPC_NOABORT     (1U << 22) 0x00400000
#define THREAD_FLAG_IPC_OPEN_RECV   (1U << 23) 0x00800000
#define THREAD_FLAG_DBG_STOPPED     (1U << 24) 0x01000000

#define DRV_TASK_PRIORITY     9 /**< max Sched priority for driver threads. */
#define SERVICE_TASK_PRIORITY 4 /**< max Sched priority for service threads. */
```

So in this particular case `flags=0x00000104 && 0xff` is giving the following info

- `0x4`, means priority 4, we know it is a TA.
- `0x100`, means `FLAG_SCHED_ENQUEUED`, the thread is ready to run.

Other examples could be:

`flags=0x005d0008`

- Priority 0x8: a driver
- 0x000d=8+4+1
- SYS_RESTART
- IPC_TX_DONE

`flags=0x00dd0009`

- Priority 0x9: a driver
- 0x000d=8+4+1
- SYS_RESTART
- IPC_TX_DONE

• IPC_RX_STARTED • 0x0050=40+10 • IPC_RX_WAIT • IPC_NOABORT Driver transmitted a (response) message to RTM, Driver is waiting to receive a new request from RTM.	• IPC_RX_STARTED • 0x00d0=80+40+10 • IPC_RX_WAIT • IPC_NOABORT • IPC_OPEN_RECV Driver transmitted a message to RTM, Driver is waiting to receive a new request from anyone.

- `ipcPartner=0x00000000` – the threadId of the communication partner for IPC send&receive.

0 – no partner	- 0xffffff – the kernel is the partner	- 0x00030001 – the RTM IPCH

- `last syscall=IPC(17)` – the last system call that this thread issued to the kernel.
  The name of the system call is decoded by tee-ps. It is not extremely useful for debug, because all of the threads have last system call = IPC.

Timing attributes

- `last modified=119846168 ms` – the last time the kernel touched this thread.
  Usually it means the time of the last system call.
  It is very helpful to reconstruct the history of last events in the system. By sorting the threads in order of last modified, we can see which threads were the last threads active in the system. Note that `tee-ps -o` option automatically sorts threads by last modified.
- `total user time=6 µs` – the time the thread has spent executing its code since the start of the thread.
  It is a helpful measure to see global execution time of a thread, also to compare it's execution time with others (not yet run, did run only a little, a lot, run more or less than another thread).
- `total sys time=9 µs` – the time this thread has spent in the microkernel, counted since the creation of this thread.
  This measure is not so helpful, since the microkernel has no long-running operations. In some cases, a thread that waits for an interrupt can have a long system time. This is especially the case for the RTM SIQH thread 102.

7 POST-MORTEM DEBUGGING

This chapter explains how the OEM can analyze bugs in the integration phase, production phase or even the end-user phase, when all components run in Release mode without debug traces.

7.1 INTRODUCTION

The integration phase starts after the development phase of the individual components: Trusted Applications, Secure Drivers, TLCs, Client Applications, Daemon, Linux kernel, Firmware. Once the TEE components work well individually, the OEM will run all the components together and do use case testing and aging tests. In case a devices crashes, the OEM can retrieve the Kinibi post-mortem debug information to analyze the issue and fix the bug.

Note that during the development phase, all components can be compiled and installed in *Debug* mode, which prints a lot of traces that help in debugging. In contrast, in the integration phase, all components run in *Release* mode and do not output a lot of traces. Hence, in case of an error it is difficult to understand what is going on.

Also, such debug is usually operated in production devices that have no hardware debugger attached and no trusted debug channel. Since the TEE is an extension of the normal world (NWd), the entire debug on production devices happens via the NWd. Note that this is a potential security problem: The TEE security objective is to protect sensitive data from the NWd. If the attacker controls the NWd (root), the attacker can also use the Kinibi debug channel. Therefore, Trustonic reduced the information in the debug channel such that no sensitive data is included.

The usual approach to debugging is to isolate the component that crashed and rerun the scenario with this and other components in Debug mode. However, eventually all the easy issues are fixed and the integrator will face issues that appear rarely and cannot be reproduced when being observed (sometimes called *Panda bugs*). The only way to find and fix such bugs is to have enough information about the system history and the current state of affairs at the moment of the crash. The Kinibi post-mortem debug information provides the necessary context to understand what events lead to the crash and what are the current open sessions and their resource consumption.

7.2 DEBUG ARCHITECTURE

The design of the post-mortem debug architecture is based on four mechanisms:

- Exposing existing systems knowledge from the Kinibi kernel driver.
- The NWd shares a buffer with the TEE and the TEE updates the buffer with debug information.

- FC_INFO fastcall retrieves limited TEE status even if TEE has crashed (See chapter 5).
- Emergency traces in the TEE even in Release mode (See Chapters 3,4,5 for `logcat` and `dmesg` analysis.).

In any case, the information is available even when the TEE has crashed. Here we describe the first two mechanisms. In both cases, the debug information is written to a memory buffer during the runtime of the TEE and can be retrieved and interpreted in case of a crash. The debug data can be retrieved in two ways: Using ramdump or by accessing them as a *root* user in Linux via the filesystem.

7.2.1 debugfs

The Kinibi kernel driver manages the access of Linux processes to the TEE. As such, it keeps a list of sessions currently running. This session list can be accessed via the Linux debugfs filesystem.

In addition, the Kinibi kernel driver knows the commands and notifications that have been send to the TEE recently. The state of the two ringbuffers for outgoing and incoming notifications can be accessed via ramdump or via the debugfs.

Finally, the Kinibi kernel driver keeps history buffers of the last commands it has sent to the TEE. There are two buffers: one for the low-level Secure Monitor Call (SMC) instructions that were issued for the TEE; and one for the high-level MobiCore Control Protocol (MCP) or Inter World Protocol (IWP) commands send to the TEE. The two buffers can be accessed via the debugfs.

7.2.2 TEE DebugSession

In case of TEE crash, the Daemon will also write the buffer of the DebugSession to the file `/data/local/tmp/debugsession`. The OEM can retrieve this file from the filesystem or access the buffer directly using ramdump.

7.3 RAMDUMP

System integrators like OEMs usually use ramdump functionality to do post-mortem debugging. It works as follows: When the Linux kernel detects a kernel panic it calls certain callbacks in various modules to create ramdump images. In most extreme case, we can image that kernel will take a snapshot of entire RAM. Such snapshot is then stored in a file, like crashdump.bin. The continuous integration system will collect this file from the device and send it for analysis. Sophisticated analysis tools then dissect the ramdump file and extract a variety of state files like traces, logs, process information, open files. Also, loading the crashdump file into a simulator and browsing the state using the binary's debug symbols is a good way to debug.

When using ramdump together with the TEE, all the TEE secure memory is a black box because the Trustzone firewalls prevent access from the NWd. However, the ramdump file will contain memory regions that the NWd has shared with the TEE. Trustonic has CMM files to extract such files from a

ramdump, using the T32 simulator and to output them in a text representation. The resulting files correspond to what could be retrieved from the device via the Linux debugfs or the mcDriverDaemon DebugSession. As such, the files retrieved from crashdump should use the same names as when retrieved using the debug interfaces. The information from these files is crucial to understand what events lead to the crash and what are the current open sessions and their resource consumption.

The following files are requested from the OEM when a device enters a TEE halt:

1. last_mcp_commands.txt
2. nq_buffer_decode.txt
3. sessions.txt
4. smc_history_decode.txt
5. teeps.txt

7.4 Trustonic Kinibi DebugFS

Linux supports a *debugfs* filesystem, usually mounted under `/sys/kernel/debug/`, but also under `/d/` on Android. See http://en.wikipedia.org/wiki/Debugfs for more details.

The Kinibi kernel driver creates the `trustonic_tee` directory in the debugfs tree that holds the following files:

Debugfs file	access rights	Contents
`active_cpu`	`write-only, root-only`	An integer entry to trigger a core switch of the SWd to the specified cpu. Depending on version, can also be used to read current core.
`crashdump`	`read-only, root-only`	A copy of the crash dump that also appears in the kernel log on TEE crash or hang. **Only created when a crash or hang occurs**.

`last_mcp_commands`	`read-only, root-only`	List of last Kinibi Legacy (MCP) commands (limited to 256).
		`PID command S-ID state result errno`
		`1928 get version 0 complete 0 0`

<table>
<tr><td></td><td></td><td>
<pre>
 1928 open session   301 complete   0   0
 1928 load token   0 complete   0   0
 1932 open session   401 complete   0   0
</pre>
</td></tr>
<tr><td>Last_iwp_commands</td><td>read-only, root-only</td><td>List of last Kinibi GP commands (limited to 256).</td></tr>
<tr><td>last_smc_commands</td><td>read-only, root-only</td><td>List of last SMC commands (limited to 256).
<pre>
CPU clock command param1   param2   param3
8918131208 -1 0x5de7e000 0x00000118 0x011800a0
8918142083 4 0x00000000 0x00000000 0x00000000
8918317917 -2 0x00000000 0x00000000 0x00000000
8918326292 3 0x00000000 0x00000000 0x00000000
8918690458 3 0x00000000 0x00000000 0x00000000
</pre>
Only created when a crash or hang occurs.</td></tr>
<tr><td>mcp_timeout</td><td>read/write, root-only</td><td>An integer to set the MCP timeout in seconds. The default value is 10. This timeout is repeated 5 times until the MCP is declared dead. In debugging sessions this feels like a watchdog bite after 50 seconds.
Example for 10 hours timeout:
<pre>
echo 36000 > /d/trustonic_tee/mcp_timeout
</pre></td></tr>
<tr><td>sessions</td><td>read-only, root-only</td><td>List of sessions known to the NWd.
<pre>
  ID type   ec state
 301 MC   0 running
 401 GP   0 running
</pre>
<table>
<tr><td>ec</td><td>Error code</td></tr>
<tr><td>MC</td><td>Legacy session</td></tr>
<tr><td>GP</td><td>GP session</td></tr>
</table></td></tr>
</table>

<table>
<tr><td></td><td></td><td>And from Kinibi-410a:
<pre>CPU clock   ID state   notif state   ec
 38233754647 401 running sent   0
           0 ff01 running idle   0</pre></td></tr>
<tr><td>structs</td><td>read-only,
root-only</td><td>Hierarchical view of the clients, sessions, buffers, mappings and MMU tables as seen inside the Kinibi kernel driver</td></tr>
<tr><td colspan="3">Content example:
<pre>client de084680 [2]: [kernel]
  cbuf dd862f80 [1]: addr dd85d000 uaddr 0 len 4096
  session de90bb00 [1]: 301 ec 0
    wsm dd862fc0: mmu de341000 cbuf dd862f80 va dd85d000 len 4096
    mmu de341000: kernel len 4096 off 0 table dd866000 type L2
client dde57000 [1]: McDaemon.FSD (1935)
  session de957e00 [2]: 401 ec 0
    wsm ddfceac0: mmu de899000 cbuf   (null) va 403ad008 len 1024000
      mmu de899000: user len 1024000 off 8 table dd864000 type L2
client dd7b3f80 [2]: INTEGRATION_Bla (2624)
  cbuf ddf97a40 [2]: addr dcd90000 uaddr 403e9000 len 4096
  session dd780200 [1]: 501 ec 0
    wsm dd6d9d40: mmu dccb6000 cbuf ddf97a40 va 403e9000 len 408
      mmu dccb6000: kernel len 40 off 0 table dcd8d000 type L2</pre>
Description:
<pre>client <struct pointer> [<number of users>]: <thread name> (<PID>)
   cbuf <struct pointer> [<number of users>]: addr <kernel virtual address> uaddr
<user-space virtual address> len <length>
   session <struct pointer> [<number of users>]: <ID> ec <exit code>
     wsm <struct pointer>: mmu <pointer to mmu> cbuf <pointer to cbuf> va <virtual
address> len <length>
       mmu <struct pointer>: <buffer type> len <length> off <offset> table <pointer
to MMU table> type L<MMU type></pre></td></tr>
<tr><td>structs_counter</td><td>read-only,
root-only</td><td>Counters of structures created and not freed, to check for memory leaks.</td></tr>
</table>

		clients: 2 cbufs: 1 sessions: 2 wsms: 2 mmus: 3
swd_debug	read/write, root-only	A boolean entry to enable or disable the SWd logging into the kernel log Note that logs themselves are still shared by Kinibi SWd image, only not displayed by TEE Linux driver.

As explained previously the main purpose of these DebugFS data is to provide execution context understanding in case of complex/low rate reproduction issues.

7.4.1 sessions.txt

The session information can be used to see what are the currently opened sessions. It shows only the sessions the NWd knows about, timestamp for last state update and only limited very high-level state: opening, running, and closing; in 410 we also show if a notification is pending for a session.

7.4.2 last_smc_commands

On a very basic level, Linux needs to issue an SMC instruction to call the TEE. The file *last_smc_commands* shows all the SMCs that Linux has 'send' to the TEE. However, only SMCs send by the Kinibi kernel driver are in this list. In addition, other SMCs or fastcalls, as send by other kernel modules, the EL2 or EL3 firmware are not in this list. This list has 1024 entries and is a circular buffer, it is maintained for debugging purpose only. Kinibi only defines a small number of SMCs and each SMC has 3 parameters.

```
#define MC_FC_INIT       MC_FC_STD32(1)    /**< Initializing FastCall. */
#define MC_FC_INFO       MC_FC_STD32(2)    /**< Info FastCall. */
#define MC_FC_MEM_TRACE  MC_FC_STD32(10)   /**< Enable SWd tracing via memory
#define MC_FC_SWAP_CPU   MC_FC_STD32(54)   /**< Change new active Core */
#define MC_SMC_N_YIELD   3                 /**< Yield to switch from NWd to SWd. */
#define MC_SMC_N_SIQ     4                 /**< SIQ to switch from NWd to SWd. */
```

When decoding the last_smc_commands, first one is often MC_FC_NWD_TRACE, to set up the trace buffer, then MC_FC_INIT to setup the communication buffer; then MC_SMC_N_SIQ to notify the TEE; then MC_FC_INFO to check if TEE is already ready. On some platforms, first SMC can be MC_FC_SWAP_CPU to do an initial fastcall to faster core.

After this initialization sequence we usually see only MC_SMC_N_SIQ and MC_SMC_N_YIELD. Here NSIQ means a notification from NWd to SWd and NYield means giving time to SWd to finish its operation. For further analysis, we can look at the parameters of NSIQ and NYield. We have 4 registers, r0-r3. R0 is the SMC or fastcall ID. R1 or Parameter p0 is mostly 0.

NYield parameters

Register	Name	Value	Description
R0	cmd	3	MC_SMC_N_YIELD
R1	p0	0	Used for return value / unused
R2	p1	4,3,2,1,0	The NWd implements a basic scheduler and sends a first yield with p1=4, then a second with 3, then 2, then 1, then 0. Afterwards NWd sends a NSIQ and then sends again a series of NYIELD.
R3	p0	0	Unused

NSIQ parameters

Register	Name	Value	Description
R0	cmd	4	MC_SMC_N_SIQ
R1	p0	0	Used for return value / unused
R2	Session ID	i.e. 0; 3;9;3f; 601	The session ID for the notification. Session ID 0 corresponds to RTM. The session ID can be 601 for the TA that has traces like 601:

R3	MCP command	1;3;4;5;9	If R2=0, this field contains the MCP command:
			`/** Open a session */`
			`MC_MCP_CMD_OPEN_SESSION = 0x01,`
			`/** Close an existing session */`
			`MC_MCP_CMD_CLOSE_SESSION = 0x03,`
			`/** Map WSM to session */`
			`MC_MCP_CMD_MAP = 0x04,`
			`/** Unmap WSM from session */`
			`MC_MCP_CMD_UNMAP = 0x05,`
			`/** Get MobiCore version information */`
			`MC_MCP_CMD_GET_MOBICORE_VERSION = 0x09,`

R3	Counter		If R2 != 0, this is a counter that is increased with each notification.

7.4.3 last_mcp_commands / last_iwp_commands

This shows the all last Kinibi legacy and GP commands, all open, close sessions and MCP_MAP, MCP_UNMAP operations…
It offers a very easy way to track what happened in the past and what the last operation were.

```
CPU clock   PID command        S-ID state    result errno UUID
30938555  1925 get version       0 complete      0     0
32027216  1925 open session    401 complete      0     0 07050501000000000000000000000020
32070518  1925 open session   ff01 complete      0     0 08050500000000000000000000000000
32103585  1925 open session    501 complete      0     0 020a0000000000000000000000000000
32174357  1925 open session    601 complete      0     0 fffffffd0000000000000000000016
34723080  1983 open session    701 complete      0     0 0706000000000000000000000000004d
34851607  1983 map            701 complete      0     0
34876406  1983 unmap          701 complete      0     0
34922585  1983 map            701 complete      0     0
34930176  1983 unmap          701 complete      0     0
34934267  1983 close session   701 complete      0     0
```

Equivalent tracking list is available for Legacy (MCP) and GP (IWP) commands.

```
CPU clock  PID   command    S-ID state    errno origin      value            UUID
792512 6395 open TA          2 complete     0   TRUSTED_APP SUCCESS  57b9fc1e645…58e106a
792513 6395 invoke command   2 complete     0   TRUSTED_APP SUCCESS
792517 6395 invoke command   2 complete     0   TRUSTED_APP SUCCESS
```

```
792521 6395 invoke command  2 complete     0   TRUSTED_APP SUCCESS
792523 6395 close session   2 complete     0   TRUSTED_APP SUCCESS
792527 6395 close session   1 complete     0   TEE         TARGET_DEAD
```

7.5 POST MORTEM DEBUG SEQUENCE

As we have seen in this chapter, in any case, "some" information are available even when the TEE has crashed. So, in case of post mortem debug the 1st overall investigation steps are always more or less similar:

1) As seen in *§5.2 Kinibi TEE Halt symptoms*, review platform `logcat` to get info on TEE crash dump. Based on field `flags` and `haltCode`, we will know the type of fatal issue:
    a. a crash;
    b. or a TEE System timeout.

2) In both case, having a way to reproduce the issue with the TEE Debug Image would be a great help and definitely the 1st thing to try.
The default debug log for Open/CloseSession will provide precious feedback on which TA/SWd driver was doing what.

3) The case of fatal crash has already been covered in previous sections and it should not be "too complex" to find back the origin of the problem.
    a. Crash dump giving UUID/IP information.
    b. Eventfully add specific debug log in TA responsible of crash to narrow down the root cause of the issue.

4) Now comes the case of TEE System timeout (or potentially low reproduction rates and random fatal crashes): In these cases, `TEE DebugFS` and `TEE DebugSession` can provide useful additional information like:
    a. From the `TEE DebugFS`, the list of SMCs called by the NWd.
    Allowing you to see roughly TEE scheduling (for example could highlight a lack of global SWd scheduling in Linux, how many SMC in 1 sec?).
    b. Also from the `TEE DebugFS`, the list of TLC/TA commands sent to understand who was doing what (for example could give context on last TA/SWd driver executed).
    This information can be cross checked with data provided by the `TEE DebugSession` and `TeePS` tool (the list of processes/threads as seen in *§6.2 TEE Kernel information* should highlight last TA scheduled)
    c. Typically, in case of a SWd driver scheduled last, it is important to review code around FIQ management and closeSession events acknowledge. Both are sensitive blocking operations that could prevent entire TEE System to respond correctly…

Even if everything is done to prevent or at least document fatal and unexpected TEE termination, it is hard to cover all potential cases. So, for sure, Trustonic Support team must be involved in complex investigation.
Ideally providing all the with following input, would definitely help the investigations:

- Exact TEE version used;
- Platform and kernel logs (`logcat` and `dmesg`);
- If able to reproduce the issue, `dmesg` of an execution with TEE Debug binary;
- Ideally an archive of `TEE DebugFS` files
  (available usually under `/sys/kernel/debug/trustonic_tee/`)
- And a dump of `TEE DebugSession`, through a ramdump or call to tee-ps,
  `./tee-ps -f -k -m -g -r -s`