TrustZone for ARM Architecture
Version 0.1a

Table of Contents

Abbreviations

BYOD	Bring Your Own Device
ARM	Advanced RISC Machine
OPTEE	Open source Portable Trusted Execution Environment
DRM	Digital Rights Management
TA	Trusted Application
CA	Client Application

1.Introduction

ARM® TrustZone® technology is a system- wide approach to security for a wide array of client and server computing platforms, including handsets, tablets, wearable devices and enterprise systems. Applications enabled by the technology are extremely varied but include payment protection technology, digital rights management, BYOD, and a host of secured enterprise solutions."

Trust zone is a set of security extensions added to ARM processors. It can run 2 operating systems. 1.Secure operating system 2.Normal operating system. Both operating system have the same capabilities and Operate in a separate memory space.

Enables a single physical processor core to execute from both the Normal world and the Secure world. Normal world components cannot access secure world resources and secure world can access normal world components.

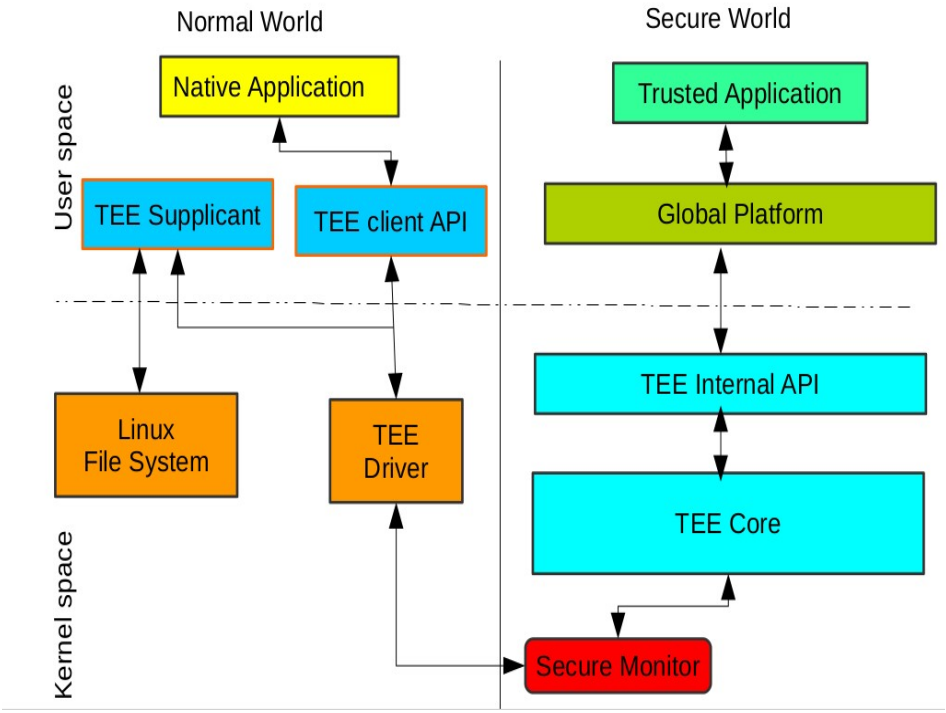

Fig1.a: Basic architecture of TrustZone

2. Architecture

Architecture of OP-TEE:

The TrustZone Hardware, where hardware extensions en-force a separation of secure and non-secure software, is more resource-efficient than the use of two separate processors. The secure world can be implemented as a full-fledged operating system, a software library, or somewhere in between. While the library design is simple, it is only suitable for occasional secure calls. Thus, the operating system model will be the focus of this discussion.

When using TrustZone to implement concurrent secure and non-secure operating systems, a general operating system, such as Linux, Android, etc., would run in the normal world and a security subsystem, such as OP-TEE or a customized Linux would run in the secure world. The functionality of the device can be described in two scenarios: first, to properly boot both operating systems; second, to provide a proper communication framework between the two.

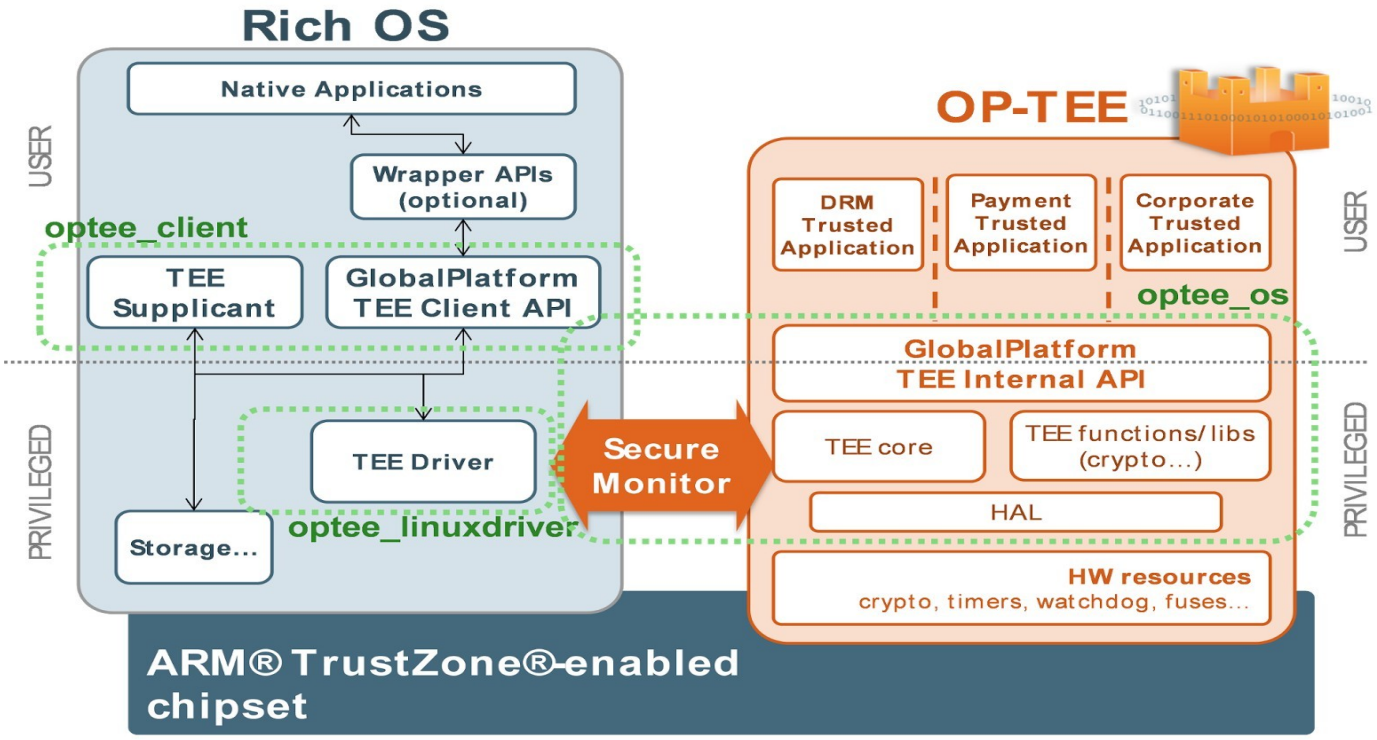

Fig2.a: Architecture of OPTEE

3. Booting sequence

Bootflow of OPTEE:

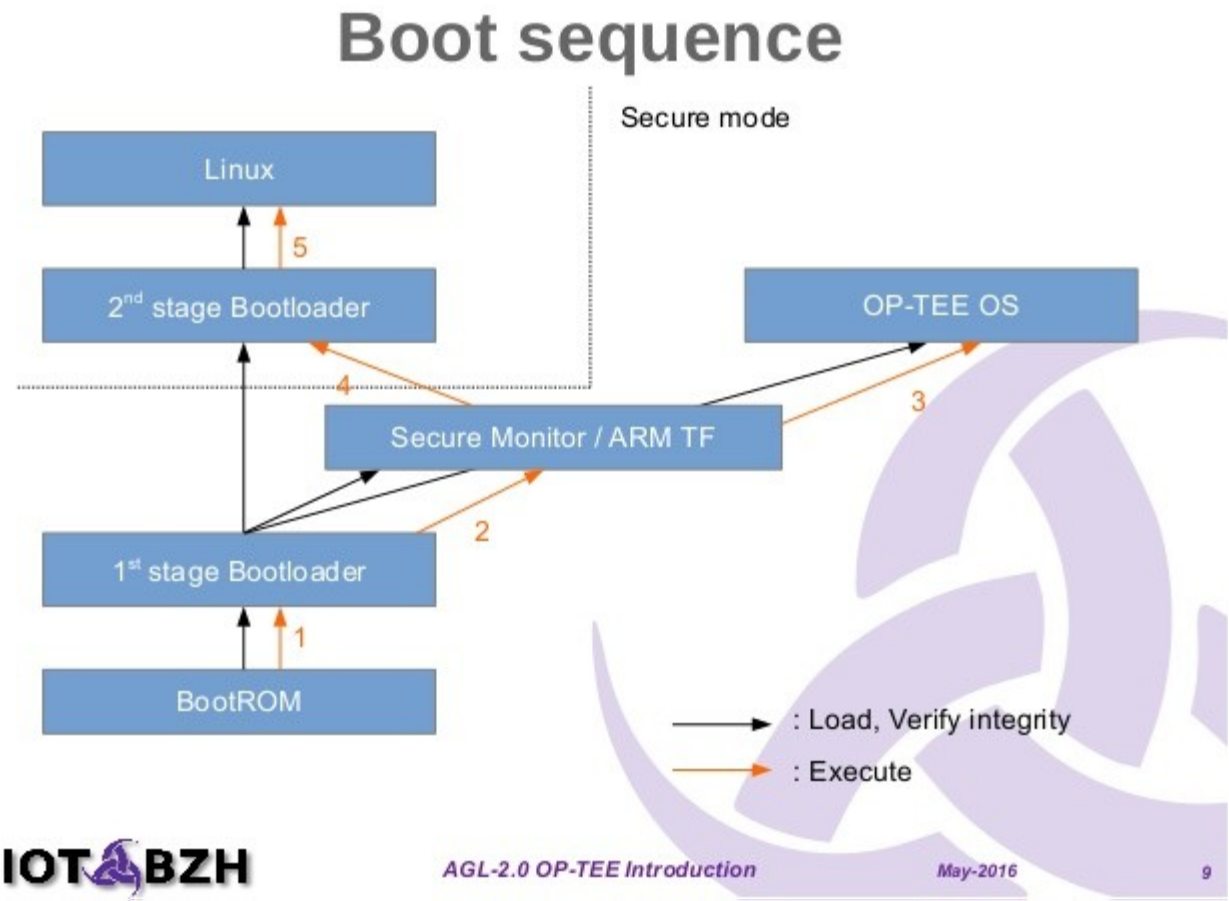

Fig3.a: Boot sequence.

4. Build steps

Step1: Install the following packages regardless of what target you will use in the end.

```
$ sudo apt-get install android-tools-adb android-tools-fastboot autoconf \
automake bc bison build-essential cscope curl device-tree-compiler \
expect flex ftp-upload gdisk iasl libattr1-dev libc6:i386 libcap-dev \
libfdt-dev libftdi-dev libglib2.0-dev libhidapi-dev libncurses5-dev \
libpixman-1-dev libssl-dev libstdc++6:i386 libtool libz1:i386 make \
mtools netcat python-crypto python-serial python-wand unzip uuid-dev \
xdg-utils xterm xz-utils zlib1g-dev
```

Step2: Install repo

Step3: Choose the manifest corresponding to the platform you intend to use. For example, if you intend to use Raspberry Pi3, then ${TARGET}.xml should be rpi3.xml.

```
$ mkdir -p $HOME/devel/optee
$ cd $HOME/devel/optee
$ repo init -u https://github.com/OP-TEE/manifest.git -m ${TARGET}.xml
$ repo sync
```

Step4: Download the toolchains

```
$ cd build
$ make toolchains
```

Step5: $ make

5. Adding New Examples

Application 1 :Added multiple functions in ta side.

Step 1 :Create your own directory e.g. $mkdir votarytech.

Step 2 :Develop all the files Android.mk , CMakeLists.txt , Makefile, and create folders ta, host.

Step 3 :Develop files Android.mk, <filename>.c, Makefile, sub.mk, user_ta_header_defines.h, include/<filename>.h inside ta folder.

Step 4 :Develop files main.c and Makefile inside host folder.

Step 5 :Generate UUID from http://www.itu.int/ITU-T/asn1/uuid.html and update it in ta/include/<filename>.h and ta/Android.mk. Define macros for each function as command IDs in ta/include/<filename.h>.

Step 6 :Define your function in ta/<filename>.c.

Step 7 :Call that functions using command id. From host/main.c file.

Step 8 :Go to terminal and Build the application using $make in build directory.

Use case:
we can call particular functionality using command id.

Application 2 :Build multiple trustzone applications with single client application.

Step 1 :Create your own directory e.g. $mkdir votarytech.

Step 2 :Develop all the files Android.mk , CMakeLists.txt , Makefile, and create folders ta, host.

Step 3 :Develop files Android.mk, <filename>.c, Makefile, sub.mk, user_ta_header_defines.h, include/<filename>.h inside ta folder.

Step 4 :Develop files main.c and Makefile inside host folder.

Step 5 :Number of Trusted applications you want to develop, Create that number of folders and Repeat from step 2 to 4.

Step 6 :Number of Trusted applications you want to develop, generate that number of UUIDs from http://www.itu.int/ITU-T/asn1/uuid.html and update it in ta/include/<filename>.h.Set the command ids i.e macros for each function in ta/include/<filename.h>.

Step 7 :By using single client application we are calling multiple ta application.

Step 8 :In host/main.c file call multiple open session apis ,multiple uuids , multiple close sessions apis,one initialize context and one finalize context.

NOTE :Make changes in makefiles as per requirement.

Eg:

```
...
TEEC_Result res;
TEEC_Context ctx;
TEEC_Session sess;
TEEC_Session sess1;
TEEC_Operation op={0};
TEEC_UUID uuid1 = TA_HELLO_WORLD_UUID;
TEEC_UUID uuid = TA_RANDOM_UUID;
```

```
uint8_t random_uuid[16] = { 0 };

...
res = TEEC_InitializeContext(NULL, &ctx);
...
res = TEEC_OpenSession(&ctx, &sess, &uuid1,TEEC_LOGIN_PUBLIC, NULL,
NULL, &err_origin);
if (res != TEEC_SUCCESS)
errx(1, "TEEC_Opensession failed with code 0x%x origin 0x%x",res, err_origin);
res = TEEC_OpenSession(&ctx, &sess1, &uuid,TEEC_LOGIN_PUBLIC, NULL,
NULL, &err_origin);
if (res != TEEC_SUCCESS)
errx(1, "TEEC_Opensession failed with code 0x%x origin 0x%x",res, err_origin);
do{
        printf("enter your choice:1.helloworld 2.Random 3.exit\n");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1:
                memset(&op, 0, sizeof(op));
                op.paramTypes = TEEC_PARAM_TYPES(TEEC_VALUE_INOUT,
                                        TEEC_NONE,TEEC_NONE, TEEC_NONE);
                op.params[0].value.a = 42;
                printf("Invoking TA to increment %d\n", op.params[0].value.a);
                res=TEEC_InvokeCommand(&sess,
                TA_HELLO_WORLD_CMD_INC_VALUE,&op,&err_origin);
`               .....
                if (res != TEEC_SUCCESS)
                errx(1, "TEEC_InvokeCommand failed with code 0x%x origin 0x
                %x",res, err_origin);
                printf("TA incremented value to %d\n", op.params[0].value.a);
                break;
                .......
        case 2:
                memset(&op, 0, sizeof(op));
                op.paramTypes=TEEC_PARAM_TYPES
                (TEEC_MEMREF_TEMP_OUTPUT,TEEC_NONE, TEEC_NONE,
                TEEC_NONE);
                op.params[0].tmpref.buffer = random_uuid;
                op.params[0].tmpref.size = sizeof(random_uuid);
                printf("Invoking TA to generate random UUID... \n");
                res=TEEC_InvokeCommand(&sess1,
                        TA_RANDOM_CMD_GENERATE,&op, &err_origin);
                if (res != TEEC_SUCCESS)
                errx(1, "TEEC_InvokeCommand failed with code 0x%x origin 0x
                        %x",res, err_origin);
                printf("TA generated UUID value = 0x");
```

```
                    for (i = 0; i < 16; i++)
                    printf("%x", random_uuid[i]);
                    printf("\n");
                    break;
                    case 3:exit(1);
            default:printf("*************invalid option ****************\n");
                    break;
                    .............
                    }
                    }while(1);

        TEEC_CloseSession(&sess);
        TEEC_CloseSession(&sess1);
        TEEC_FinalizeContext(&ctx);
        return 0;
        }
```

Step 9 **:**Go to terminal and Build the application using $make in build directory.

Use case :
If one client applications want to use multiple trusted application in single trusted os.

Application 3: Build multiple client applications using single trusted applications.

Step 1 :Create your own directory e.g. $mkdir votarytech.

Step 2 :Develop all the files Android.mk , CMakeLists.txt , Makefile, and create folders ta, host.

Step 3 :Develop files Android.mk, <filename>.c, Makefile, sub.mk, user_ta_header_defines.h, include/<filename>.h inside ta folder.

Step 4 :Develop files main.c and Makefile inside host folder.

Step 5 :Number of Trusted applications you want to develop, Create that number of folders and Repeat from step 2 to 4.

Step 6 :Generate a UUIDs from http://www.itu.int/ITU-T/asn1/uuid.html and update it in ta/include/<filename>.h.Set the command ids for each function in ta/include/<filename.h>.

Step7 **:**Change Makefile to add number of host directories as shown below.

```
export V?=0
# If _HOST or _TA specific compilers are not specified, then use
CROSS_COMPILE
HOST_CROSS_COMPILE ?= $(CROSS_COMPILE)
TA_CROSS_COMPILE ?= $(CROSS_COMPILE)
.PHONY: all
all:
$(MAKE) -C host CROSS_COMPILE="$(HOST_CROSS_COMPILE)" –no-
builtin-variables
$(MAKE) -C ta CROSS_COMPILE="$(TA_CROSS_COMPILE)" LDFLAGS=""
$(MAKE) -C host1 CROSS_COMPILE="$(HOST_CROSS_COMPILE)" –no-
builtin-variables
.PHONY: clean
```

```
clean:
$(MAKE) -C host clean
$(MAKE) -C host1 clean
$(MAKE) -C ta clean
```

Step 8 **:** Call Initialize_contexts, open sessions,invoke_command, close session,Single UUID and finalize_context from multiple CA files.

Step 9 **:**Go to terminal and Build the application using $make in build directory.

6. Flashing steps

Step1: Partitioning Sd-card

```
$ fdisk /dev/sdx   # where sdx is the name of your sd-card
> p           # prints partition table
> d           # repeat until all partitions are deleted
> n           # create a new partition
> p           # create primary
> 1           # make it the first partition
> <enter>      # use the default sector
> +32M         # create a boot partition with 32MB of space
> n           # create rootfs partition
> p
> 2
> <enter>
> <enter>      # fill the remaining disk, adjust size to fit your needs
> t           # change partition type
> 1           # select first partition
> e           # use type 'e' (FAT16)
> a           # make partition bootable
> 1           # select first partition
> p           # double check everything looks right
> w           # write partition table to disk.
```

Step2: loading images into boot

run the following as root

```
$ mkfs.vfat -F16 -n BOOT /dev/sdx1
$ mkdir -p /media/boot
$ mount /dev/sdx1 /media/boot
$ cd /media
$ gunzip -cd /home/<home directory>/trustzone/build/../out-br/images/rootfs.cpio.gz
| sudo  cpio -idmv "boot/*"
$ umount boot
```

Step3: Loading images into root

run the following as root

```
$ mkfs.ext4 -L rootfs /dev/sdx2
$ mkdir -p /media/rootfs
$ mount /dev/sdx2 /media/rootfs
$ cd rootfs
$ gunzip -cd /home/<home directory>/trustzone/build/../out-br/images/rootfs.cpio.gz
| sudo  cpio -idmv
$ rm -rf /media/rootfs/boot/*
$ cd .. && umount rootfs
```

7. Errors

1. uboot.env file is missing:

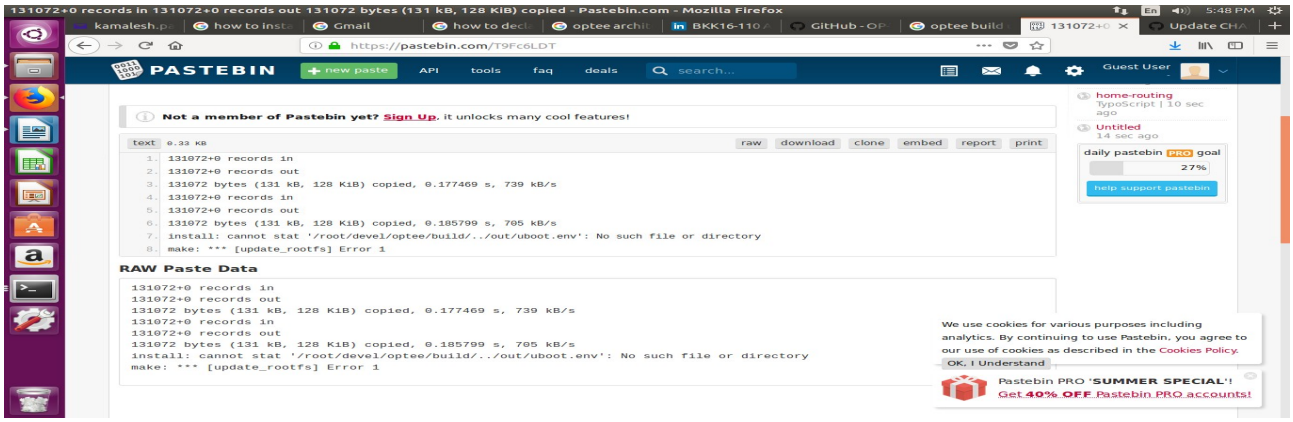

Fig7.a: Error showing image

Solution: Copy uboot.env.txt from /home/<home directory>/trustzone/build/rpi3/firmware to /<home directory>/trustzone/out and remname it as uboot.env.

2. Toolchain issue and solutions:

while downloading toolchain if you encounter following issues then follow below steps to solve it.

ISSUE:

* cd qemu/build
* make toolchain

error log:

Downloading gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf ...
tar:/home/<homedirectory>/Q8TEE/build/../toolchains/gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf.tar.xz: Cannot open: No such file or directory
tar: Error is not recoverable: exiting now
toolchain.mk:35: recipe for target 'aarch32' failed
make: *** [aarch32] Error 2

Solution :

This error is due to the broken download link and you can download it from below link
https://releases.linaro.org/components/toolchain/binaries/6.2-2016.11/arm-linux-gnueabihf/
then download the gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf.tar.xz
go to:
https://releases.linaro.org/components/toolchain/binaries/6.2-2016.11/aarch64-linux-gnu
then download gcc-linaro-6.2.1-2016.11-x86_64_aarch64-linux-gnu.tar.xz
After downloading copy both files to toolchains directory
then run
cd qemu/build
make toolchains

8. Debugging OP-TEE

Debugging OP-TEE
In this document we would like to describe how to debug OP-TEE.

1. QEMU
To debug OP-TEE using QEMU you could use `gdb` as the main debugger. Using this setup will also make it possible to use some GDB Front End tools, if you don't feel comfortable using a command line debugging tool.

1.1 Prerequisites
Since there are inter-dependencies between the gits used when building OP-TEE, we recommend that you have been following the guide as described in [README.md].
```bash
# Root folder for the project
$HOME/devel/optee
```
1.2 Download gdb for ARM
If you've followed the instructions until here, then you should have the toolchains already in bash
$HOME/devel/optee/toolchains
```
## 1.3 Scripts
A few helper scripts that makes life easier.
Start by creating `$HOME/.gdbinit` and add:
```
set print array on
set print pretty on
define optee
	handle SIGTRAP noprint nostop pass
	symbol-file $HOME/devel/optee/optee_os/out/arm/core/tee.elf
	target remote localhost:1234
end
document optee
	Loads and setup the binary (tee.elf) for OP-TEE and also connects to the QEMU
	remote.
end
```
Now you are good to go for doing debugging using command line.

## 1.4 Debug
Start QEMU according to the instructions in [QEMU ARMv7-A], **however**, do not start the emulation, i.e. do not type the `c` command in QEMU. The main reason for not doing so is because you cannot set the breakpoints on secure side when kernel has booted up. And then in another terminal start gdb using following command-
$HOME/devel/optee/toolchains/aarch64/bin/arm-linux-gnueabihf-gdb -q
To connect to the remote and to load the `tee.elf`, simply type:
```

```
(gdb) optee
SIGTRAP is used by the debugger.
Are you sure you want to change it? (y or n) [answered Y; input not from
terminal]
0x00000000 in ?? ()
```
Now it is time to set the breakpoints. For example
```
(gdb) b tee_entry_std
Breakpoint 1 at 0x7df0c7be: file core/arch/arm/tee/entry_std.c, line 268.
```
and then start the execution by writing the continue command in gdb.
```
(gdb) c
Continuing.
```
When the driver has been loaded and you start using OP-TEE the breakpoint will trigger, which will look something like this:
```
Breakpoint 1, tee_entry_std (smc_args=0x7df6ff98 <stack_thread+8216>)
 at core/arch/arm/tee/entry_std.c:268
268     struct optee_msg_arg *arg = NULL;
(gdb)
```
2. Debugging TA
Assumptions:
1. You already know how to setup `gdb` from the previous chapter;
2.. We use the Hello World TA ([`linaro-swg/optee_examples/hello_world`](https://github.com/linaro- swg/optee_examples/));
3. `pwd` is `…/hello_world/ta`

First, we need to find out the LMA (Load Memory Address) of Hello World TA. To do that run `objdump` utility (you might need to use platform specific one from the toolchain of the platform you are building for):
```bash
$ objdump -h 8aaaf200-2450-11e4-abe2-0002a5d5c51b.elf
```
The result will look something like this:
```
Sections:
Idx  Name       Size      VMA              LMA              File off  Algn
  0  .ta_head   00000020  0000000000000000  0000000000000000  00010000  2**3
               CONTENTS, ALLOC, LOAD, DATA
  1  .text      0000c614  0000000000000020  0000000000000020  00010020  2**3
               CONTENTS, ALLOC, LOAD, READONLY, CODE
  2  .rodata    00002382  000000000000c638  000000000000c638  0001c638  2**3
```

Here, we are interested in the LMA (Load Memory Address) of `.text` section. The effective virtual address of the TA is computed at runtime from a load base address and the LMA of the `.text` section in the TA ELF file. We will see below how to get this virtual memory load base address. Now, you can run gdb and connect to QEMU session.
```
(gdb) optee
SIGTRAP is used by the debugger.
Are you sure you want to change it? (y or n) [answered Y; input not from
terminal]
0x00000000 in ?? ()
```
We have to set a breakpoint inside the OP-TEE OS. Since we want to debug the TA, we can set a breakpoint at a stage where the OS has fully loaded and mapped the TA so we can access its whole memory. The best candidate seems `thread_enter_user_mode`. It is called when OP-TEE OS executes a TA entrypoint.
```
(gdb) b tee_thread_enter_user_mode
Breakpoint 1 at 0xe106974: file core/arch/arm/kernel/thread.c, line 1100.
```
then continue the execution of vm:
```
(gdb) c
Continuing.
```
Now you can run Hello World CA (Client Application) from Normal World FVP terminal:
```bash
# / optee_hello_world
```
`gdb` will hit the breakpoint at `tee_thread_enter_user_mode`, which will look something like this:
```
Breakpoint 1, thread_enter_user_mode
(a0=0, a1=409104, a2=1073747840, a3=0, user_sp=1073747840, entry_func=1073803664, is_32bit=false, exit_status0=0xe163b00, exit_status1=0xe163b04)
at core/arch/arm/kernel/thread.c:1105
1105            if (!get_spsr(is_32bit, entry_func, &spsr)) {
(gdb)
```
In the Secure World terminal window you should be able to see something like this:
```
DEBUG: [0x0] TEE-CORE:tee_ta_init_pseudo_ta_session:259: Lookup for pseudo TA 8aaaf200-2450-11e4-abe2-0002a5d5c51b
DEBUG: [0x0] TEE-CORE:tee_ta_init_user_ta_session:610: Load user TA 8aaaf200-2450-11e4-abe2-0002a5d5c51b
DEBUG: [0x0] TEE-CORE:ta_load:316: ELF load address 0x40001000
```

Here, we are interested in ELF load address value, in this case it is `0x40001000`.This is the actual address of Hello World TA loaded by OP-TEE OS. To be able to debug Hello World TA you have to add symbols of the application into `gdb` using `add-symbol-file [file] [address]` command, were `[address]` is
ELF load address + LMA of `.text` section. In this case, it is `0x40001000 + 0x20`
Let’s add those symbols:
```
(gdb) add-symbol-file ./8aaaf200-2450-11e4-abe2-0002a5d5c51b.elf 0x40001020
```
You should see something like:
```
add symbol table from file "./8aaaf200-2450-11e4-abe2-0002a5d5c51b.elf" at .text_addr = 0x40001020
```
Let's setup a breakpoint inside Hello World TA, for example:
```
(gdb) b TA_InvokeCommandEntryPoint
Breakpoint 2 at 0x400028e0: file hello_world_ta.c, line 135.
```
Here one can remove the optee core breakpoint (`thread_enter_user_mode`) and continue the execution of vm:
```
(gdb) del 1
(gdb) c
Continuing.
```
Voila! `gdb` should hit the breakpoint inside Hello World TA and you should be able to see something like this:
```
Breakpoint 2, TA_InvokeCommandEntryPoint
(sess_ctx=0x0 <ta_head>, cmd_id=cmd_id@entry=0, param_types=param_types@entry=3, params=params@entry=0x40000f40) at hello_world_ta.c:135
135             switch (cmd_id) {
```

9. Package manager

Steps for Adding Libraries and Packages in OPTEE

The below steps demonstrates how to add gstreamer library in your OPTEE

1. cd ~/qemu/build
2. vi common.mk
   at line 230 add bellow two lines
   @echo "BR2_PACKAGE_GSTREAMER1=y" >> ../out-br/extra.conf
   @echo "BR2_PACKAGE_GSTREAMER1_MM=y" >> ../out-br/extra.conf
3. This will add the required packages for adding gstreamer library in build and out-br
   and then run make

For adding any other packages like c++ in optee you can follow the above steps
but before that make sure that package is available in qemu/buildroot/package

then check the config.in and find the package name macro to add in common.mk

---