OP-TEE - How to debug

OP-TEE - How to debug

Stable: 24.06.2020 - 13:00 / Revision: 24.06.2020 - 13:00

Contents

1 Purpose

This article explains how to debug the OP-TEE secure world binaries.

This debug information is specifically linked to the CPU secure state (Arm® TrustZone®).

The OP-TEE secure world binaries include OP-TEE core (privilege firmware) and OP-TEE trusted applications and libraries (user space context):

- There are two main ways to debug OP-TEE core: using embedded traces, or using JTAG/SWD to access the secure world.
  The focus here is on the solution integrated in OpenSTLinux: debug over GDB (ST-LINK or JTAG/SWD based).
- The OP-TEE trusted applications and libraries provide debug support relying on embedded traces only.
  Debugging the secure userland binaries through JTAG/SWD resources is not recommended, and OP-TEE does not provide any means to embed a GDB server in the secure world.

This article focuses on OP-TEE debug.
Refer to STM32MP1 Platform trace and debug environment overview article for more generic information.

2 Debugging OP-TEE core

2.1 OP-TEE Version number

The starting point for debugging OP-TEE core is to identify the OP-TEE version embedded in the target. A version identifier is displayed on the console with the following format:

```
I/TC: OP-TEE version:<tag> #<buildcount> <date> <arch>
```

that is:

```
I/TC: OP-TEE version: openstlinux-19-01-11-10-g56ef3b0 #1 Wed Jan 30 09:12:56 UTC 2019
arm
```

2.2 Embedded assertions

OP-TEE core can embed debug assertions that panic the system when the tested condition is not met. Embedded assertions are implemented using the **assert()** function, that is, in the following code snippet, the system panics if **argument1** is negative, while the function returns the decremented value of the input argument:

```
static int increment_argument(int argument)
{
   assert(argument > 0);
   return argument - 1;
}
```

Assertions are embedded (or not) depending on the configuration directive **CFG_TEE_CORE_DEBUG={n|y}** when the OP-TEE core is built.

2.3 Debug with traces

OP-TEE provides trace message support and a configurable trace level build directive **CFG_TEE_CORE_LOG_LEVEL={0|1|2|3|4}**. This directive defines the trace levels that are embedded inside the firmware and output through the OP-TEE console. A low value reduces the memory footprint of the firmware and increases its runtime performance.

Value	Name	Description with related macros
0	-	All trace messages are disabled
1	Error trace level	Only non-tagged and error trace messages are embedded: MSG(), MSG_RAW(), EMSG(), EMSG_RAW()
	Info trace	

Value	Name	Description with related macros
2	level	+ info trace messages: IMSG(), IMSG_RAW()
3	Debug trace level	+ debug trace messages: DMSG(), DMSG_RAW()
4	Flow trace level	+ flow trace messages: FMSG(), FMSG_RAW()

Concurrent enabling of all debug supports is not possible due to the internal memory size constraint. If required, change some DMSG() into IMSG().

Traces and errors are available on the console defined in the chosen node of the device tree by the stdout-path property:

```
chosen {
        stdout-path = "serial0:115200n8";
};
```

More information about OP-TEE build and update is available in the STM32MP15 OP-TEE article.

2.4 Stack unwind support

The OP-TEE core can trace the execution backtrace when its panics. The execution backtrace allows analysis of the execution call sequence that led to the panic. The OP-TEE OS provides tools to analyse such backtraces based on the OP-TEE core ELF file generated at build time.

Backtrace unwind is embedded (or not) depending on the configuration directive **CFG_UNWIND={y|n}** Note that backtrace unwind increases the size of the OP-TEE core firmware. When the OP-TEE core executes from a small secure RAM, enabling stack unwind penalizes the OP-TEE core performance.

More information is available from the OP-TEE abort-dumps documentation[1].

2.5 Debug with GDB

The Debug OpenSTLinux BSP Components with GDB article describes how to set up the GDB / OpenOCD environment. OP-TEE can be debugged through JTAG/SWD using an ST-LINK or the JTAG/SWD output, depending on the target board.

Note that when the OP-TEE core executes in a small secure memory with the support of its pager (case build directive **CFG_WITH_PAGER=y**), use of hardware breakpoints rather than software breakpoints is highly recommended. Since most of the OP-TEE core instructions are dynamically loaded into the small secure memory, a loaded software breakpoint is likely to be discarded when the OP-TEE pager wipes memory content to load other OP-TEE core pages.

When OP-TEE executes in the large main memory (case build directive **CFG_WITH_PAGER** is disabled), all OP-TEE core resources are resident. In this case, hardware breakpoints as well as software breakpoints can be used without any issues.

2.5.1 Debug boot sequence

Load symbols to the target offset:

```
(gdb) add-symbol-file <path_to_build_folder>/tee.elf <load_address>
```

OP-TEE load address is available from the generated **tee-init_load_addr.txt** file.
It can also be found in the generated **tee.map** file:

```
...
Linker script and memory map

                0x000000002ffc0000                . = 0x2ffc0000
                0x0000000000000001                ASSERT (0x1, text start should align
to 32bytes)
                0x000000002ffc0000                __text_start = .
                0x000000002ffc0000                __flatmap_unpg_rx_start =
((__text_start / 0x1000) * 0x1000)

.text           0x000000002ffc0000     0xc538
 *(SORT_BY_ALIGNMENT(.text._start))
 .text._start   0x000000002ffc0000       0x98 out/stm32mp157c-ev1/core/arch/arm/kernel
/generic_entry_a32.o
                0x000000002ffc0000                _start  -> OP-TEE Load address
...
```

In this example, the OP-TEE load address is 0x2ffc0000.

All OP-TEE core symbols can be loaded:

```
(gdb) add-symbol-file <path_to_build_folder>/tee.elf 0x2ffc0000
```

Thanks to the Wrapper_for_FSBL_images, you will be able to debug the initial boot sequence. Once the board starts and waits into the FSBL debug wrapper, a hardware breakpoint can be set at the OP-TEE core entry point.

```
(gdb) hb _start
```

OP-TEE symbols maybe override the FSBL once, you cannot load both TF-A and OP-TEE symbols at the same time.

2.5.2 Debugging during runtime execution

Once U-Boot or the Linux kernel is running, secure memory or regions cannot be accessed, but it is possible to break by setting a hardware breakpoint on OP-TEE service handler. GDB breaks once it has switched into the secure world and reached the break instruction. Once halted, GDB can access secure resources as peripheral interfaces or memories. For example, to break into U-Boot use the following GDB instructions:

```
(gdb) hb stm32_sip_service
(gdb) continue
```

On the first service call occurrence GDB breaks into the **stm32_sip_service()** entry in the OP-TEE core.

3 Debugging OP-TEE trusted applications

3.1 Embedded assertions

OP-TEE trusted applications can embed assertions as well as the OP-TEE core, as described above.
Assertions in trusted applications are embedded on the configuration directive **CFG_TEE_CORE_DEBUG={y|n}** when the OP-TEE OS package is built.

3.2 Debug with traces

OP-TEE trusted applications can embed trace messages using the same macros as the OP-TEE core (**EMSG()** and similar functions described above).
The build directive that sets the trace level for a trusted application is **CFG_TEE_TA_LOG_LEVEL={0|1|2|3|4}**.
The level values match those described for **CFG_TEE_CORE_LOG_LEVEL** in the section above.

3.3 Stack unwind support

When an OP-TEE trusted application panics, it may output backtrace messages on the OP-TEE console through the OP-TEE core.
These backtrace messages are generated by the OP-TEE core, which must be built with **CFG_UNWIND=y** as described in the section above.

4 References

- https://github.com/OP-TEE/optee_os/blob/master/documentation/abort_dumps.rst

Central processing unit

Open Portable Trusted Execution Environment

debug and test protocol, named from the Joint Test Action Group that developed it

Serial Wire Debug

GNU dedugger, a portable debugger that runs on many Unix-like systems

Trusted Execution Environment

Operating System

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today forvolatile random-accesssemiconductor memories.)

First Stage Boot Loader

Trusted Firmware for Arm Cortex-A

Trusted Application