ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

BSEC device tree configuration

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

Contents

1. BSEC device tree configuration ... 3
2. BSEC internal peripheral ... 10
3. Device tree ... 16
4. NVMEM overview ... 21
5. STM32CubeMX ... 29
6. TF-A overview ... 32

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

A quality version of this page, approved on *5 November 2020*, was based off this revision.

Contents

1 Article purpose 4
2 DT bindings documentation 5
3 DT configuration 6
3.1 DT configuration (STM32 level) 6
3.2 DT configuration (board level) 6
3.2.1 STM32MP1 BSEC node append 6
3.2.2 STM32MP1 BSEC node append (bootloader specific) 7
3.2.3 STM32MP1 driver node append 7
3.2.4 STM32MP1 nvmem_layout node (bootloader specific) 8
4 How to configure the DT using STM32CubeMX 9
5 References 10

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

1 Article purpose

This article explains how to configure BSEC at boot time.

This article describes the BSEC configuration performed using the device tree mechanism, which provides a hardware description of the BSEC peripheral.

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

2 DT bindings documentation

Generic information about NVMEM is available in the NVMEM overview.

The following binding-related documentation explains how to write device tree files for BSEC:

- TF-A: *tf-a/docs/devicetree/bindings/soc/st,stm32-romem.txt"*[1]
- Linux®BSEC devicetree bindings: Documentation/devicetree/bindings/nvmem/st,stm32-romem.txt[2]
- Linux® generic NVMEM devicetree bindings: Documentation/devicetree/bindings/nvmem/nvmem.yaml[3] and Documentation /devicetree/bindings/nvmem/nvmem-consumer.yaml[4]

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

3 DT configuration

This hardware description is a combination of the **STM32 microprocessor** device tree files (*.dtsi* extension) and **board** device tree files (*.dts* extension). See the Device tree for an explanation of the device-tree file split.

STM32CubeMX can be used to generate the board device tree. Refer to How to configure the DT using STM32CubeMX for more details.

3.1 DT configuration (STM32 level)

The STM32MP1 BSEC node is located in the file *stm32mp151.dtsi*[5] (see Device tree for further explanation).

```
/ {
...
        soc {
...
                bsec: nvmem@5c005000 {
                        compatible = "st,stm32mp15-bsec";
                        reg = <0x5c005000 0x400>;
                        #address-cells = <1>;
                        #size-cells = <1>;

                        part_number_otp: part_number_otp@4 {
                                reg = <0x4 0x1>;
                        };
                        ts_cal1: calib@5c {
                                reg = <0x5c 0x2>;
                        };
                        ts_cal2: calib@5e {
                                reg = <0x5e 0x2>;
                        };
                };
...
        };
...
};
```

Please refer to the NVMEM overview for the bindings common with the Linux® kernel.

3.2 DT configuration (board level)

3.2.1 STM32MP1 BSEC node append

The board definition in the device tree may include some additional board-specific OTP declarations:

```
&bsec {
        board_id: board_id@ec {
                reg = <0xec 0x4>;
                st,non-secure-otp;
        };
};
```

With only 32 lower NVMEM 32-bit data words, the software needs to manage exceptions in order to allow some upper OTPs to be accessed by the non-secure world, through secure world services for very specific needs. The user can add an OTP declaration in the device tree, using the "st,non-secure-otp" property, with a 32-bit length granularity (that is, 4 bytes).

3.2.2 STM32MP1 BSEC node append (bootloader specific)

The bootloader-specific STM32MP1 BSEC node append data is located in the file *stm32mp151.dtsi* [6] for TF-A (see Device tree for further explanation).
This completes NVMEM data providers, for bootloader-specific purposes only, either for a driver, or the platform itself.

```
bsec: nvmem@5c005000 {
        compatible = "st,stm32mp15-bsec";
        reg = <0x5c005000 0x400>;
        #address-cells = <1>;
        #size-cells = <1>;

        cfg0_otp: cfg0_otp@0 {
                reg = <0x0 0x1>;
        };
        part_number_otp: part_number_otp@4 {
                reg = <0x4 0x1>;
        };
        monotonic_otp: monotonic_otp@10 {
                reg = <0x10 0x4>;
        };
        nand_otp: nand_otp@24 {
                reg = <0x24 0x4>;
        };
        uid_otp: uid_otp@34 {
                reg = <0x34 0xc>;
        };
        package_otp: package_otp@40 {
                reg = <0x40 0x4>;
        };
        hw2_otp: hw2_otp@48 {
                reg = <0x48 0x4>;
        };
        ts_cal1: calib@5c {
                reg = <0x5c 0x2>;
        };
        ts_cal2: calib@5e {
                reg = <0x5e 0x2>;
        };
        pkh_otp: pkh_otp@60 {
                reg = <0x60 0x20>;
        };
        mac_addr: mac_addr@e4 {
                reg = <0xe4 0x8>;
                st,non-secure-otp;
        };
};
```

Please see the "st,non-secure-otp" definition in the previous section above. No more spare field declaration here.

3.2.3 STM32MP1 driver node append

The driver can directly consume NVMEM data cells, as described in NVMEM overview.
The CPU0 device is a good example, with a dedicated OTP containing part number information.
The device node is located in the *stm32mp151.dtsi*[5] file.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

```
cpu0: cpu@0 {
        compatible = "arm,cortex-a7";
        device_type = "cpu";
        reg = <0>;
        clocks = <&scmi0_clk CK_SCMI0_MPU>;
        clock-names = "cpu";
        operating-points-v2 = <&cpu0_opp_table>;
        nvmem-cells = <&part_number_otp>;
        nvmem-cell-names = "part_number";
        #cooling-cells = <2>;
};
```

With these nvmem-cells / nvmem-cell-names properties, the CPU0 device can easily find the OTP number, in order to access part number information.

3.2.4 STM32MP1 nvmem_layout node (bootloader specific)

The STM32MP1 nvmem_layout node gathers all NVMEM platform-dependent layout information, including OTP names and phandles, in order to allow easy access for data consumers, using pre-defined string in the nvmem-cell-names property.

```
nvmem_layout: nvmem_layout@0 {
        compatible = "st,stm32mp1-nvmem-layout";
        nvmem-cells = <&cfg0_otp>,
                      <&part_number_otp>,
                      <&monotonic_otp>,
                      <&nand_otp>,
                      <&uid_otp>,
                      <&package_otp>,
                      <&hw2_otp>;

        nvmem-cell-names = "cfg0_otp",
                           "part_number_otp",
                           "monotonic_otp",
                           "uid_otp",
                           "nand_otp",
                           "package_otp",
                           "hw2_otp";
};
```

With this new node, the platform can easily find the OTP numbers, in order to access all the necessary information.

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

4 How to configure the DT using STM32CubeMX

The STM32CubeMX tool can be used to configure the STM32MPU device and get the corresponding platform configuration device tree files.
STM32CubeMX may not support all the properties described in the documents listed in DT bindings documentation above. If so, the tool inserts **user sections** in the generated device tree. These sections can then be edited to add some properties that are preserved from one generation to another. Refer to the STM32CubeMX user manual for further information.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

5 References

Please refer to the following links for additional information:

- docs/devicetree/bindings/soc/st,stm32-romem.txt TF-A BSEC binding information file
- Documentation/devicetree/bindings/nvmem/st,stm32-romem.txt
- Documentation/devicetree/bindings/nvmem/nvmem.yaml
- Documentation/devicetree/bindings/nvmem/nvmem-consumer.yaml
- 5.05.1 arch/arm/boot/dts/stm32mp151.dtsi : STM32MP151 Linux kernel device tree files
- fdts/stm32mp151.dtsi STM32MP151 TF-A device tree files

Device Tree

Boot and Security and OTP control

Linux® is a registered trademark of Linus Torvalds.

One Time Programmed

Microprocessor Unit

Trusted Firmware for Arm® Cortex®-A

Stable: 24.09.2019 - 14:06 / Revision: 24.09.2019 - 07:57

A quality version of this page, approved on *24 September 2019*, was based off this revision.

Contents

1 Article purpose 11
2 Peripheral overview 12
2.1 Features 12
2.2 Security support 12
3 Peripheral usage and associated software 13
3.1 Boot time 13
3.2 Runtime 13
3.2.1 Overview 13
3.2.2 Software frameworks 13
3.2.3 Peripheral configuration 13
3.2.4 Peripheral assignment 13
4 How to go further 15
5 References 16

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

1 Article purpose

The purpose of this article is to

- briefly introduce the BSEC peripheral and its main features
- indicate the level of security supported by this hardware block
- explain how each instance can be allocated to the three runtime contexts and linked to the corresponding software components
- explain, when necessary, how to configure the BSEC peripheral.

2 Peripheral overview

The **BSEC** peripheral is used to control an OTP (one time programmable) fuse box, used for on-chip non-volatile storage for device configuration and security parameters.

2.1 Features

Refer to STM32MP15 reference manuals for the complete list of features, and to the software components, introduced below, to see which features are implemented.

2.2 Security support

The BSEC is a **secure** peripheral.

3 Peripheral usage and associated software

3.1 Boot time

The BSEC is configured at boot time to set up platform security.

3.2 Runtime

3.2.1 Overview

The BSEC instance is a system peripheral and is controlled by the Arm®Cortex®-A7 secure:

- BSEC lower OTP access can be made available to the Arm®Cortex®-A7 non-secure.
- Upper OTP access can be managed as exceptions (in Trusted Boot Chain only, using TF-A), via "secure monitor calls", managed by TF-A or by OP-TEE. Please refer to BSEC device tree configuration for more details.

3.2.2 Software frameworks

Domain	Peripheral	Software components			Comment
OP-TEE	Linux	STM32Cube			
Security	BSEC	OP-TEE BSEC driver	Linux NVMEM framework		

3.2.3 Peripheral configuration

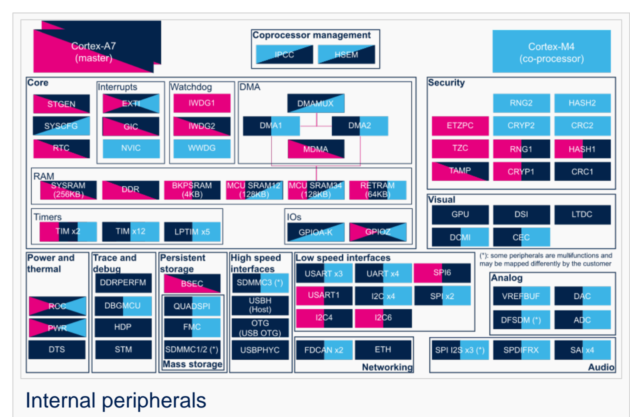
The configuration is based on Device tree, please refer to BSEC device tree configuration article.
It can be applied by the firmware running in a secure context, done in TF-A or in OP-TEE.
It can also be configured by Linux® kernel, please refer to NVMEM overview article.

3.2.4 Peripheral assignment

Check boxes illustrate the possible peripheral allocations supported by STM32 MPU Embedded Software:

- means that the peripheral can be assigned () to the given runtime context.
- is used for system peripherals that cannot be unchecked because they are statically connected in the device.

Refer to How to assign an internal peripheral to a runtime context for more information on how to assign peripherals manually or via STM32CubeMX.
The present chapter describes STMicroelectronics recommendations or choice of implementation. Additional possiblities might be described in STM32MP15 reference manuals
.



Internal peripherals

Domain	Peripheral	Runtime allocation				Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Security	BSEC	BSEC				

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

4 How to go further

5 References

Boot and Security and OTP control

One Time Programmed

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Cortex®

Open Portable Trusted Execution Environment

Linux® is a registered trademark of Linus Torvalds.

Stable: 19.03.2021 - 08:52 / Revision: 19.03.2021 - 08:49

A quality version of this page, approved on *19 March 2021*, was based off this revision.

Contents

1 Purpose 17
1.1 Source files 17
1.2 Bindings 17
1.3 Build 17
1.4 Tools 18
2 STM32 19
3 How to go further 20
4 References 21

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

1 Purpose

The objective of this chapter is to give general information about the device tree.
An extract of the **device tree specification**[1] explains it as follows:

"A device tree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent. ... Rather than hard coding every detail of a device into an operating system, many aspect of the hardware can be described in a data structure that is passed to the operating system at boot time."

In other words, a device tree describes the hardware that can not be located by probing. For more information, please refer to the device tree specification[1]

1.1 Source files

- **.dts:** The device tree source (DTS). This format is a textual representation of a device tree in a form that can be processed by DTC (Device Tree Compiler) into a binary device tree in the form expected by software components: Linux® Kernel, U-Boot and TF-A.

- **.dtsi:** Source files that can be included from a DTS file.

1.2 Bindings

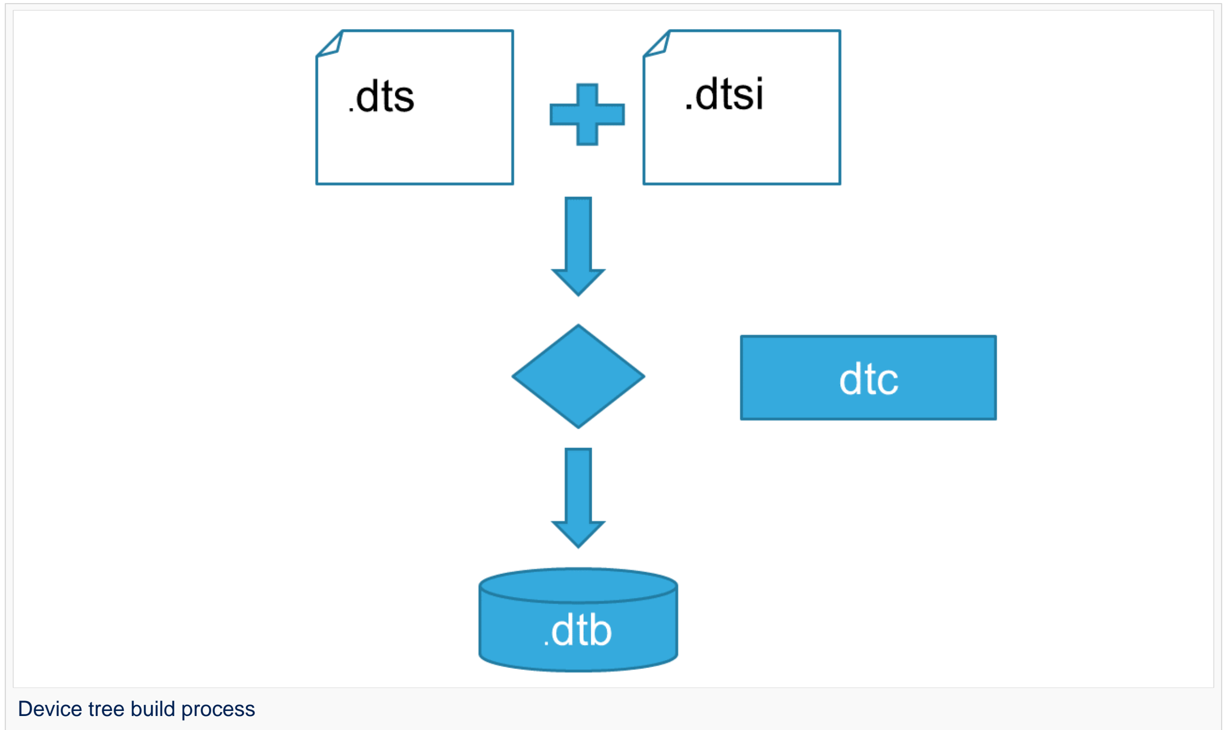
The device tree data structures and properties are named **bindings**. Those bindings are described in:

- The Device tree specification[1] for generic bindings.
- The software component documentations:
  - Linux® Kernel: Linux kernel device tree bindings
  - U-Boot: doc/device-tree-bindings/
  - TF-A: TF-A device tree bindings

1.3 Build

- A tool named DTC (Device Tree Compiler) allows compiling the DTS sources into a binary.
- input file: the *.dts* file described in section above.
- output file: the *.dtb* file described in section above.
- More information are available in DTC manual[2].

- DTC source code is located here[3]. DTC tool is also available directly in particular software



Device tree build process

components: **Linux Kernel, U-Boot, TF-A ....** For those components, the device tree building is directly integrated in the component build process.

If dts files use some defines, dts files should be preprocessed before being compiled by DTC.

1.4 Tools

The device tree compiler offers also some tools:

- **fdtdump:** Print a readable version of a flattened device tree file (dtb)
- **fdtget:** Read properties from a device tree
- **fdtput:** Write properties to a device tree
- ...

There are several ways to get those tools:

- In the device tree compiler project source code[3]
- Directly in software components: **Kernel, u-boot, tf-a ...**
- Available in Debian package[4]

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 STM32

For STM32MP1, the device tree is used by three software components: Linux® kernel, U-Boot and TF-A.

The device tree is part of the OpenSTLinux distribution. It can also be generated by STM32CubeMX tool.

To have more information about the device tree usage on STM32MP1 (how the device tree source files are split, how to find the device tree source files per software components, how is STM32CubeMX generating the device tree ...) see STM32MP15 device tree page.

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

3 How to go further

- Device Tree for STM32MP [5]
- Device Tree Reference[6] - eLinux.org
- Device Tree usage[7] - eLinux.org

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

4 References

- [1.01.11.2] https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.2(latest) ,Device tree specification
- https://git.kernel.org/pub/scm/utils/dtc/dtc.git/tree/Documentation/manual.txt(master) ,DTC manual
- [3.03.1] https://git.kernel.org/pub/scm/utils/dtc/dtc.git(master) ,DTC source code
- https://packages.debian.org/search?keywords=device-tree-compiler(master) ,DTC debian package
- https://www.youtube.com/watch?v=a9CZ1Uk3OYQ, Device Tree for STM32MP
- Device Tree Reference, eLinux.org
- Device Tree Usage, eLinux.org

Device Tree Source (in software context) or Digital Temperature Sensor (in peripheral context)

Linux® is a registered trademark of Linus Torvalds.

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Trusted Firmware for Arm® Cortex®-A

Stable: 04.11.2020 - 14:14 / Revision: 04.11.2020 - 14:04

A quality version of this page, approved on *4 November 2020*, was based off this revision.

This article introduces how NVMEM Linux® framework manages BSEC OTP data and how to read/write from/to it.

Contents

1 Framework purpose 22
2 System overview 23
2.1 Component description 23
2.2 API description 24
3 Configuration 25
3.1 Kernel configuration 25
3.2 Device tree configuration 25
4 How to use the framework 26
4.1 How to use NVMEM with sysfs interface 26
4.1.1 How to list NVMEM devices 26
4.1.2 How to read BSEC lower OTPs using NVMEM 26
4.1.3 How to read BSEC upper OTPs using NVMEM 26
4.1.4 How to write BSEC OTPs using NVMEM 26
5 How to trace and debug the framework 28
5.1 How to trace 28
6 References 29

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

1 Framework purpose

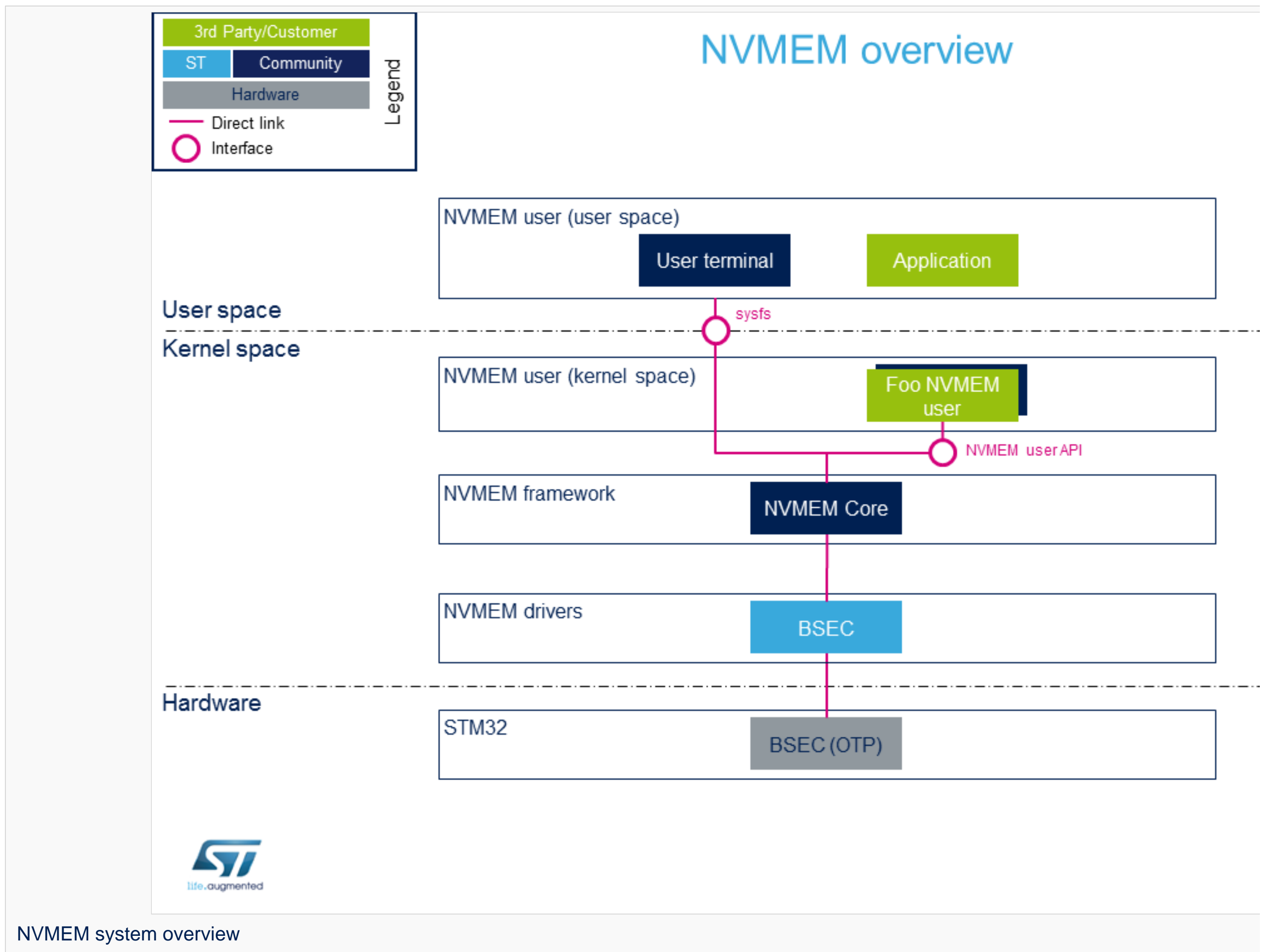
The NVMEM Linux® framework provides a generic interface for the device **non-volatile memory data** such as:

- OTP (one-time programmable) fuses
- EEPROM

It offers kernel space and user space interfaces to read and/or write data such as analog calibration data or MAC address.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 System overview



NVMEM system overview

2.1 Component description

- **NVMEM user** (user space)

The user can use the NVMEM sysfs interface, from a user terminal or a custom application, to read/write data from/to NVMEM device(s) from user space.

- **NVMEM user** (kernel space)

User drivers can use the NVMEM API to read/write data from/to NVMEM device(s) from kernel space (such as the analog calibration data used by an ADC driver).

- **NVMEM framework** (kernel space)

The NVMEM core provides sysfs interface and NVMEM API. They can be used to implement NVMEM user and NVMEM controller drivers.

- **NVMEM drivers** (kernel space)

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Provider drivers such as BSEC Linux® driver that exposes OTP data to the core.

- **NVMEM hardware**

NVMEM controller(s) such as the *BSEC internal peripheral*[1]

2.2 API description

The NVMEM kernel documentation[2] describes:

- Kernel space API for NVMEM **providers** and NVMEM **consumers**.
- Userspace binary interface (sysfs).

See also *sysfs-bus-nvmem*[3]ABI documentation.

3 Configuration

3.1 Kernel configuration

Activate NVMEM framework in the kernel configuration through the Linux® menuconfig tool, Menuconfig or how to configure kernel (CONFIG_NVMEM=y):

```
Device Drivers  --->
   [*] NVMEM Support  --->
      <*>   STMicroelectronics STM32 factory-programmed memory support
```

3.2 Device tree configuration

The NVMEM data device tree bindings describe:

- The location of non-volatile memory data
- The NVMEM data providers[4]
- The NVMEM data consumers[5]

The *BSEC internal peripheral*[1] device tree bindings are explained in BSEC device tree configuration article.

4 How to use the framework

4.1 How to use NVMEM with sysfs interface

4.1.1 How to list NVMEM devices

The available NVMEM devices can be listed in sysfs:

```
# Example to list nvmem devices
Board $> ls /sys/bus/nvmem/devices/
stm32-romem0
```

The data content of an NVMEM device can be dumped to a binary file, and then displayed.

4.1.2 How to read BSEC lower OTPs using NVMEM

The **32 lower OTPs** can be read from non-secure when using either:

- the trusted boot chain (using TF-A)
- the basic boot chain (using U-Boot SPL)

```
# Example to read lower nvmem data content
Board $> dd if=/sys/bus/nvmem/devices/stm32-romem0/nvmem of=/tmp/file bs=4 count=32
# Example to display nvmem data content
Board $> hexdump -C -v /tmp/file
```

4.1.3 How to read BSEC upper OTPs using NVMEM

Only the 32 lower OTPs can be accessed when using the basic boot chain, as it doesn't implement secure services (CONFIG_HAVE_ARM_SMCCC). So this section concerns only the trusted boot chain (using TF-A) as SMC feature is available.

Default behavior for upper OTPS is normally restricted to security. If user needs more than the 32 lower OTPs, there is an exception management explained in BSEC device tree configuration.

It is then possible to access to some upper NVMEM information.

```
# Example to read the MAC address from upper OTP area, using secure services:
Board $> dd if=/sys/bus/nvmem/devices/stm32-romem0/nvmem of=/tmp/file skip=57 bs=4
count=2 status=none
Board $> hexdump -C -v /tmp/file
```

A dedicated chapter of the reference manual describes the OTP mapping.

4.1.4 How to write BSEC OTPs using NVMEM

The below examples show how to write data to an NVMEM device. This may cause unrecoverable damage to the STM32 device (for example when writing to an OTP area)

Note that lower OTPs are using 2:1 redundancy, so they can be written bit per bit, whereas upper OTPs only support one time 32-bit programming.

Whatever the boot chain, the full lower NVMEM data content can be written as follows (if we suppose it has been previously read as described above, and updated directly in /tmp/file):

```
# Example to write lower nvmem data content
Board $> dd if=/tmp/file of=/sys/bus/nvmem/devices/stm32-romem0/nvmem bs=4 count=32
```

Only on Trusted boot chain, and under the condition the device tree authorizes it, an upper NVMEM data can be written.
Example of 32-bit data word writing (filling it with ones) in OTP n°95:

```
# Create a 4 bytes length file filled with ones, e.g. 0xffffffff)
# Then, write it (32-bits, e.g. 4bytes) to OTP data 95
Board $> dd if=/dev/zero count=1 bs=4 | tr '\000' '\377' > file
Board $> dd if=file bs=4 seek=95 of=/sys/bus/nvmem/devices/stm32-romem0/nvmem
```

When a new OTP value has been written using this SYSFS interface, it may be necessary to reboot the board before reading it back. The OTP value can't be read directly after a write because the OTP value is read in a shadow area not directly in the OTP area.

5 How to trace and debug the framework

5.1 How to trace

Ftrace can be used to trace the NVMEM framework:

```
Board $> cd /sys/kernel/debug/tracing
Board $> cat available_filter_functions | grep nvmem             # Show available filter
functions
rtc_nvmem_register
rtc_nvmem_unregister
nvmem_reg_read
bin_attr_nvmem_read
...
```

Enable the kernel function tracer, then start using nvmem and display the result:

```
Board $> echo function > current_tracer
Board $> echo "*nvmem*" > set_ftrace_filter                      # Trace all nvmem filter
functions
Board $> echo 1 > tracing_on                                     # start ftrace
Board $> hexdump -C -v /sys/bus/nvmem/devices/stm32-romem0/nvmem # dump nvmem
00000000  17 00 00 00 01 80 00 00  00 00 00 00 00 00 00 00  |................|
...
Board $> echo 0 > tracing_on                                     # stop ftrace
Board $> cat trace
# tracer: function
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
         hexdump-478   [000] ....   423.502278: bin_attr_nvmem_read <-sysfs_kf_bin_read
         hexdump-478   [000] ....   423.502290: nvmem_reg_read <-bin_attr_nvmem_read
         hexdump-478   [000] ....   423.515804: bin_attr_nvmem_read <-sysfs_kf_bin_read
```

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

6 References

- 1.01.1 BSEC internal peripheral
- Documentation/driver-api/nvmem.rst , NVMEM subsytem kernel documentation
- Documentation/ABI/stable/sysfs-bus-nvmem , NVMEM ABI documentation
- Documentation/devicetree/bindings/nvmem/nvmem.yaml , NVMEM device tree bindings
- Documentation/devicetree/bindings/nvmem/nvmem-consumer.yaml , NVMEM consumer device tree bindings

Linux® is a registered trademark of Linus Torvalds.

Boot and Security and OTP control

One Time Programmed

Electrically-erasable programmable read-only memory

media access control address (https://en.wikipedia.org/wiki/MAC_address)

System File System (See https://en.wikipedia.org/wiki/Sysfs for more details)

Application programming interface

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

Application binary interface. (In computer software, an application binary interface (ABI) describes the low-level interface between a computer program and the operating system or another program.)

secure monitor call (SMC) calling convention

Secure Monitor Call

Central processing unit

Stable: 23.09.2020 - 13:22 / Revision: 12.06.2020 - 13:25

A quality version of this page, approved on *23 September 2020*, was based off this revision.

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

1 STM32CubeMX overview

This article describes STM32CubeMX, an official STMicroelectronics graphical software configuration tool.

The STM32CubeMX application helps developers to use the STM32 by means of a user interface, and guides the user through to the initial configuration of a firmware project.

It provides the means to:

- configure pin assignments, the clock tree, or internal peripherals
- simulate the power consumption of the resulting project
- configure and tune DDR parameters
- generate HAL initialization code for Cortex-M4
- generate the Device Tree for a Linux kernel, TF-A and U-Boot firmware for Cortex-A7

It uses a rich library of data from the STM32 microcontroller portfolio.

The application is intended to ease the initial development phase by helping developers to select the best product in terms of features and power.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 STM32CubeMX main features

- Peripheral and middleware parameters

  Presents options specific to each supported software component
- Peripheral assignment to processors

  Allows assignment of each peripheral to Cortex-A Secure, Cortex-A Non-Secure, or Cortex-M processors
- Power consumption calculator

  Uses a database of typical values to estimate power consumption, DMIPS, and battery life
- Code generation

  Makes code regeneration possible, while keeping user code intact
- Pinout configuration

  Enables peripherals to be chosen for use, and assigns GPIO and alternate functions to pins
- Clock tree initialization

  Chooses the oscillator and sets the PLL and clock dividers
- DDR tunning tool

  Ensures the configuration, testing, and tuning of the MPU DDR parameters. Using U-Boot-SPL Embedded Software.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

3 How to get STM32CubeMX

Please, refer to the following link STM32CubeMX to find STM32CubeMX, the Release Note, the User Manual and the product specification.

Doubledata rate (memory domain)

Hardware Abstraction Layer

Cortex®

Linux® is a registered trademark of Linus Torvalds.

Trusted Firmware for Arm® Cortex®-A

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

General-Purpose Input/Output (A realization of open ended transmission between devices on an embedded level. These pins available on a processor can be programmed to be used to either accept input or provide output to external devices depending on user desires and applications requirements.)

Microprocessor Unit

Stable: 22.04.2021 - 11:23 / Revision: 09.04.2021 - 13:17

A quality version of this page, approved on *22 April 2021*, was based off this revision.

Contents

1 Trusted Firmware-A 33
2 Architecture 34
3 Boot loader stages 36
3.1 BL1 36
3.2 BL2 36
3.2.1 FIP 36
3.2.2 Firmware Configuration 36
3.2.3 Authentication 37
3.3 BL32 37
4 References 38

1 Trusted Firmware-A

Trusted Firmware-A is a reference implementation of secure-world software provided by Arm®. It was first designed for Armv8-A platforms, and has been adapted to be used on Armv7-A platforms by STMicroelectronics. Trusted Firmware-A is part of the Trusted Firmware project that is an open governance community project hosted by Linaro.[1]

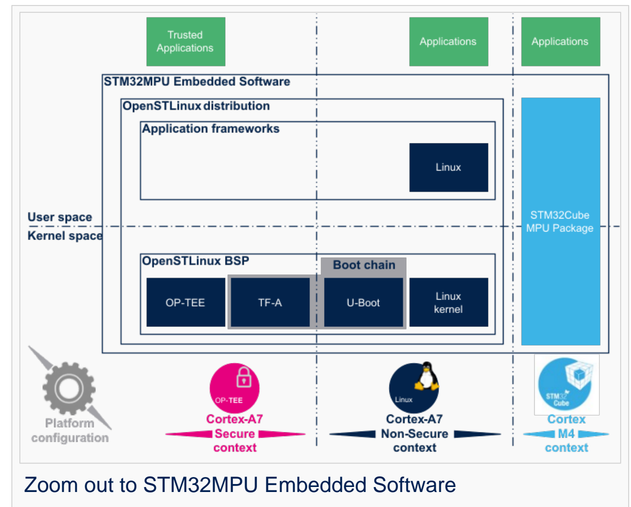
It is used as the first-stage boot loader (FSBL) on STM32 MPU platforms when using the trusted boot chain.

The code is open source, under a BSD-3-Clause license, and can be found on Linaro project page [2], including an up-to-date documentation about Trusted Firmware-A implementation [3].

Trusted Firmware-A also implements a set of features with various Arm interface standards:

- The power state coordination interface (PSCI) [4]
- SMC calling convention [5]
- System control and management interface [6]

Trusted Firmware-A is usually shortened to TF-A.



Zoom out to STM32MPU Embedded Software

2 Architecture

The global architecture of TF-A is explained in the Trusted Firmware-A design [7] document.

TF-A is divided into several binaries, each with a dedicated main role.
For 32-bit Arm processors (AArch32), the trusted boot is divided into four stages (in order of execution):

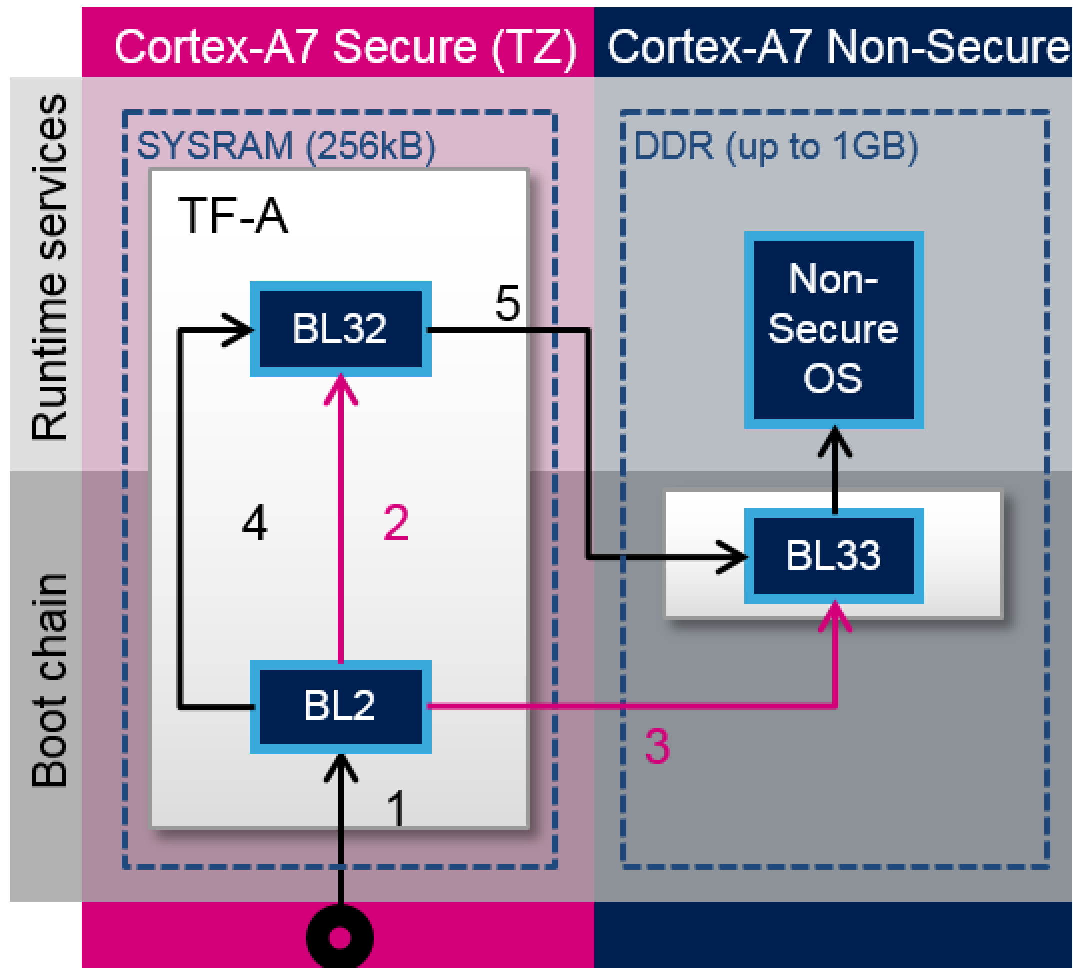
- Boot loader stage 1 (BL1) application processor trusted ROM
- Boot loader stage 2 (BL2) trusted boot firmware
- Boot loader stage 3-2 (BL32) runtime software
- Boot loader stage 3-3 (BL33) non-trusted firmware

BL1, BL2 and BL32 are parts of TF-A.

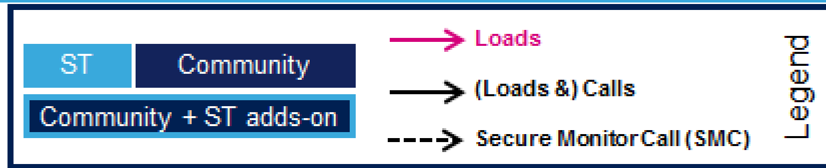
Because STM32 MPU platforms uses a dedicated ROM code, the BL1 boot stage is then removed. ROM code expects the BL2 to run at EL3 execution level. This mode is selected when the BL2_AT_EL3 build flag is enabled.

BL33 is outside of TF-A. This is the first non-secure code loaded by TF-A. During the boot sequence, this is the secondary stage boot loader (SSBL). For STM32 MPU platforms, the SSBL is U-Boot by default.

TF-A can manage its configuration with a device tree. In the BL2 stage, it is a reduced version of the Linux kernel one, with only the required devices used during boot. It can be configured with STM32CubeMX.



ST Restriced – Subject to Non-Disclosure Agreement – Do not copy



TF-A loading steps:

1. ROM code loads and calls BL2
2. BL2 loads BL32
3. BL2 loads BL33
4. BL2 calls BL32
5. BL32 calls BL33

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

3 Boot loader stages

3.1 BL1

BL1 is the first stage executed, and is designed to act as ROM code; it is loaded and executed in internal RAM. It is not used for the STM32 MPU. As the STM32 MPU has its own proprietary ROM code, this part can be removed and BL2 is then the first TF-A binary to be executed.

3.2 BL2

BL2 is in charge of loading the next-stage images (secure and non secure). To achieve this role, BL2 has to initialize all the required peripherals.

- System components: clocks, DDR, ...
- Security components: Firewall
- Storage

BL2 offers different features to load and authenticate images.

At the end of its execution, after having loaded BL32 and the next boot stage (BL33), BL2 jumps to BL32.

3.2.1 FIP

The Firmware Image Package (FIP)[8] is a TF-A archive binary that encapsulates bootloader images into a single archive. It can also contain other data such as certificates that are required to complete the boot process. A dedicated driver drivers/io/io_fip.c able to read data from this package is part of the TF-A BL2.
FIP uses a specific layout based on a table of contents followed by payload data. It is synchronized between the driver and the host creation tool: Fiptool tools/fiptool/fiptool.c . This host tool is able to create a package, get info from the package, update, unpack or remove data in this package.

3.2.2 Firmware Configuration

The Firmware Configuration Framework (FCONF)[9] is a way to offer more flexibility in the firmware. It is used to provide most of the platform-specific data that were previously hard coded inside the firmware. This framework uses device tree (one or multiple) that are passed to the firmware during load processing. BL2 uses it to describe the chain of trust and the images list to be loaded.
Thanks to device tree usage, the configuration becomes dynamic at boot time. The current implementation uses the following device tree as framework entry:

- FW_CONFIG - The firmware configuration file. Hold properties shared across all BLx images. An example is the **dtb-registry** node, which contains the information about other binaries configuration (load-address, size, image_id).
- HW_CONFIG - The hardware configuration file. Can be shared by all Boot Loader stages and also by the Normal World Rich OS.
- TB_FW_CONFIG - Trusted Boot Firmware configuration file. Shared between BL1 and BL2.
- SOC_FW_CONFIG - SoC Firmware configuration file. Used by BL31.
- TOS_FW_CONFIG - Trusted OS Firmware configuration file. Used by Trusted OS (BL32).
- NT_FW_CONFIG - Non Trusted Firmware configuration file. Used by Non-trusted firmware (BL33).

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

3.2.3 Authentication

TF-A BL2 implements an authentication framework that uses a defined Chain of Trust (CoT) based on Arm TBBR[10] requirement to achieve a secure boot. The authentication is enabled as soon as the TRUSTED_BOARD_BOOT flag is defined. TF-A BL2 implements this CoT which is based on a Root of Trust Public Key (ROTPK). The CoT relies on a public key infrastructure generating self-signed certificate (following X509 v3 standard [11]). There is **no Certificate Authority (CA)** because the CoT is not established by verifying the validity of a certificate's issuer.

Different keys are used for this CoT:

- Root of trust key - The private part of this key is used to sign the BL2 content certificate and the trusted key certificate. The public part is the ROTPK.
- Trusted world key - The private part is used to sign the key certificates corresponding to the secure world images (SCP_BL2 , BL31 and BL32). The public part is stored in one of the extension fields in the trusted world certificate.
- Non-trusted world key - The private part is used to sign the key certificate corresponding to the non secure world image (BL33). The public part is stored in one of the extension fields in the trusted world certificate.
- BL3X keys - For each of SCP_BL2, BL31, BL32 and BL33, the private part is used to sign the content certificate for the BL3X image. The public part is stored in one of the extension fields in the corresponding key certificate.

The certificates used in this CoT could be Key certificate or Content certificate.

- BL2 content certificate - It is self-signed with the private part of the ROT key. It contains a hash of the BL2 image.
- Trusted key certificate - It is self-signed with the private part of the ROT key. It contains the public part of the trusted world key and the public part of the non-trusted world key.
- SCP_BL2 key certificate - It is self-signed with the trusted world key. It contains the public part of the SCP_BL2 key.
- SCP_BL2 content certificate - It is self-signed with the SCP_BL2 key. It contains a hash of the SCP_BL2 image.
- BL31 key certificate - It is self-signed with the trusted world key. It contains the public part of the BL31 key.
- BL31 content certificate - It is self-signed with the BL31 key. It contains a hash of the BL31 image.
- BL32 key certificate - It is self-signed with the trusted world key. It contains the public part of the BL32 key.
- BL32 content certificate - It is self-signed with the BL32 key. It contains a hash of the BL32 image.
- BL33 key certificate - It is self-signed with the non-trusted world key. It contains the public part of the BL33 key.
- BL33 content certificate - It is self-signed with the BL33 key. It contains a hash of the BL33 image.

3.3 BL32

BL32 provides runtime secure services.

On Armv7 architecture, the BL32 must embed a Secure Monitor as it will be executed in the same privilege level (PL1-SVC Secure). TF-A provides a minimal monitor implementation: SP-MIN. It is described in the TF-A functionality list [3] as: "A minimal AArch32 Secure Payload (SP-MIN) to demonstrate PSCI [4] library integration with AArch32 EL3 Runtime Software."

This minimal implementation can be replaced with a trusted OS or trusted environment execution (TEE), such as OP-TEE that also embeds a secure monitor on Armv7. Both solutions (SP-MIN or OP-TEE) are supported by STMicroelectronics for STM32MP15.

BL32 acts as a secure monitor and thus provides secure services to non-secure OSs. These services are called by non-secure software with secure monitor calls [5].

This code is in charge of standard service calls, like PSCI [4] or SCMI [6].

It also provides STMicroelectronics proprietary services to access secure peripherals (with secure access control).

4 References

- https://www.trustedfirmware.org/
- https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git
- 3.03.1 https://trustedfirmware-a.readthedocs.io/en/latest/
- 4.04.14.2 ARM Power State Coordination Interface
- 5.05.1 SMC Calling Convention (SMCCC)
- 6.06.1 Arm System Control and Management Interface
- https://trustedfirmware-a.readthedocs.io/en/latest/design/index.html
- https://trustedfirmware-a.readthedocs.io/en/latest/design/firmware-design.html?highlight=FIP#firmware-image-package-fip
- https://trustedfirmware-a.readthedocs.io/en/latest/components/fconf/index.html?highlight=FCONF
- https://trustedfirmware-a.readthedocs.io/en/latest/design/trusted-board-boot.html
- https://tools.ietf.org/rfc/rfc5280.txt

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

First Stage Boot Loader

Microprocessor Unit

Power State Coordination Interface

Secure Monitor Call

Trusted Firmware for Arm® Cortex®-A

Boot Loader stage 1

Read Only Memory

Boot Loader stage 2

Boot Loader stage 3-2

Boot Loader stage 3-3

Second Stage Boot Loader

Linux® is a registered trademark of Linus Torvalds.

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 for volatile random-acces ssemiconductor memories.)

Doubledata rate (memory domain)

Firmware Image Package is a packaging format used by TF-A

Firmware Configuration Framework used by TF-A - NEW

Operating System

Chain of Trust

Secure coprocessor

Secure Payload minimal

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Trusted Execution Environment

Open Portable Trusted Execution Environment

System control and management interface