NVMEM overview

Contents

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

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.

2 System overview

Error: Image is invalid or non-existent.

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)

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

Information

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
```

Information

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

4.1.4 How to write BSEC OTPs using NVMEM

Warning

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)

Information

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
```

Information

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
```

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

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