Category:Crypto

Contents

1. Category:Crypto 3
2. CRC device tree configuration 4
3. CRYP device tree configuration 9
4. Crypto API overview 14
5. HASH device tree configuration 23

Stable: 17.06.2020 - 15:26 / Revision: 16.01.2020 - 07:58

Category:Crypto

Stable: 17.06.2020 - 10:20 | Revision: 16.01.2020 - 07:38

A quality version of this page, approved on *17 June 2020*, was based off this revision.

This category groups together all articles related to the Linux® **cryptography** software framework.

It is recommended to first read the Crypto API overview article.

Pages in category "Crypto"

The following 4 pages are in this category, out of 4 total.

- Crypto API overview
- CRC device tree configuration
- CRYP device tree configuration
- HASH device tree configuration

Stable: 13.05.2020 - 08:24 / Revision: 13.05.2020 - 08:20

A quality version of this page, approved on *13 May 2020*, was based off this revision.

Contents

1 Article purpose 5
2 DT bindings documentation 6
3 DT configuration 7
3.1 DT configuration (STM32 level) 7
3.2 DT configuration (board level) 7
3.3 DT configuration examples 7
4 How to configure the DT using STM32CubeMX 8
5 References 9

1 Article purpose

The purpose of this article is to explain how to configure the *CRC*[1] **when the peripheral is assigned to Linux®OS**.

The configuration is performed using the **device tree mechanism**[2].

The Device tree provides a hardware description of the *CRC*[1], used by STM32 *CRC Linux driver*.

If the peripheral is assigned to another execution context, refer to How to assign an internal peripheral to a runtime context article for guidelines on peripheral assignment and configuration.

2 DT bindings documentation

The *CRC* [1] is represented by the *STM32 CRC device tree bindings*[3]

3 DT configuration

This hardware description is a combination of STM32 microprocessor and board device tree files. See Device tree for explanations on device tree file split.

The **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 CRC node is declared in stm32mp151.dtsi[4]. It provides the hardware registers base address and the clock.

```
crc1: crc@58009000 {
        compatible = "st,stm32f7-crc";
        reg = <0x58009000 0x400>;
        clocks = <&rcc CRC1>;
        status = "disabled";
};
```

Warning

This device tree part is related to STM32 microprocessors. It should be kept as is, without being modified by the end-user.

3.2 DT configuration (board level)

This part is used to enable the CRC used on a board. This is done by setting the **status** property to **okay**.

3.3 DT configuration examples

```
&crc1 {
        status = "okay";
};
```

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.
The STM32CubeMX may not support all the properties described in the above DT bindings documentation paragraph. If so, the tool inserts **user sections** in the generated device tree. These sections can then be edited to add some properties and they are preserved from one generation to another. Refer to STM32CubeMX user manual for further information.

5 References

Please refer to the following links for additional information:

- 1.01.11.2 CRC internal peripheral
- Device tree
- Documentation/devicetree/bindings/crypto/st,stm32-crc.txt
- STM32MP151 device tree file

Cyclic redundancy check calculation unit

Linux® is a registered trademark of Linus Torvalds.

Operating System

Device Tree

Stable: 13.05.2020 - 08:30 / Revision: 13.05.2020 - 08:28

A quality version of this page, approved on *13 May 2020*, was based off this revision.

Contents

1 Article purpose 10
2 DT bindings documentation 11
3 DT configuration 12
3.1 DT configuration (STM32 level) 12
3.2 DT configuration (board level) 12
3.3 DT configuration examples 12
4 How to configure the DT using STM32CubeMX 13
5 References 14

1 Article purpose

The purpose of this article is to explain how to configure the *CRYP*[1] **when the peripheral is assigned to Linux®OS**.

The configuration is performed using the **device tree mechanism**[2].

The Device tree provides a hardware description of the *CRYP*[1], used by STM32 *CRYP Linux driver*.

If the peripheral is assigned to another execution context, refer to How to assign an internal peripheral to a runtime context article for guidelines on peripheral assignment and configuration.

2 DT bindings documentation

The *CRYP* [1] is represented by the *STM32 CRYP device tree bindings*[3].

3 DT configuration

This hardware description is a combination of STM32 microprocessor and board device tree files. See Device tree for explanations on device tree file split.

The **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 CRYP node is declared in stm32mp15xc.dtsi[4] and stm32mp15xf.dtsi[5]. It provides the hardware registers base address, clock, interrupt and reset.

```
cryp1: cryp@54001000 {
        compatible = "st,stm32mp1-cryp";
        reg = <0x54001000 0x400>;
        interrupts = <GIC_SPI 79 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&scmi0_clk CK_SCMI0_CRYP1>;
        resets = <&scmi0_reset RST_SCMI0_CRYP1>;
        status = "disabled";
};
```

Warning

This device tree part is related to STM32 microprocessors. It should be kept as is, without being modified by the end-user.

3.2 DT configuration (board level)

This part is used to enable the CRYP used on a board. This is done by setting the **status** property to **okay**.

3.3 DT configuration examples

```
&cryp1 {
        status = "okay";
};
```

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.
The STM32CubeMX may not support all the properties described in the above DT bindings documentation paragraph. If so, the tool inserts **user sections** in the generated device tree. These sections can then be edited to add some properties and they are preserved from one generation to another. Refer to STM32CubeMX user manual for further information.

5 References

Please refer to the following links for additional information:

- 1.01.11.2 CRYP internal peripheral
- Device tree
- Documentation/devicetree/bindings/crypto/st,stm32-cryp.txt
- STM32MP157C device tree file
- STM32MP157F device tree file

Cryptographic processor

Linux® is a registered trademark of Linus Torvalds.

Operating System

Device Tree

Generic Interrupt Controller

Serial Peripheral Interface

Stable: 19.10.2020 - 09:54 / Revision: 19.10.2020 - 09:52

A quality version of this page, approved on *19 October 2020*, was based off this revision.

The Crypto API is a cryptography framework in the Linux® kernel. It is dedicated to the parts of the kernel that deal with cryptography, such as IPsec and dm-crypt.

Contents

1 Framework purpose 15
2 System overview 16
2.1 Description of the components 16
2.2 API description 17
3 Configuration 18
3.1 Kernal configuration 18
3.2 Devicetree configuration 18
4 How to use the Crypto API framework 19
5 Use cases 20
6 How to trace and debug the framework 21
6.1 How to monitor 21
6.2 How to trace 21
6.3 How to debug 21
7 Generic source code location 22
8 References 23

1 Framework purpose

The purpose of this article is to introduce the Crypto API framework:

- general information
- main component/stakeholders
- how to use the Crypto API
- use cases

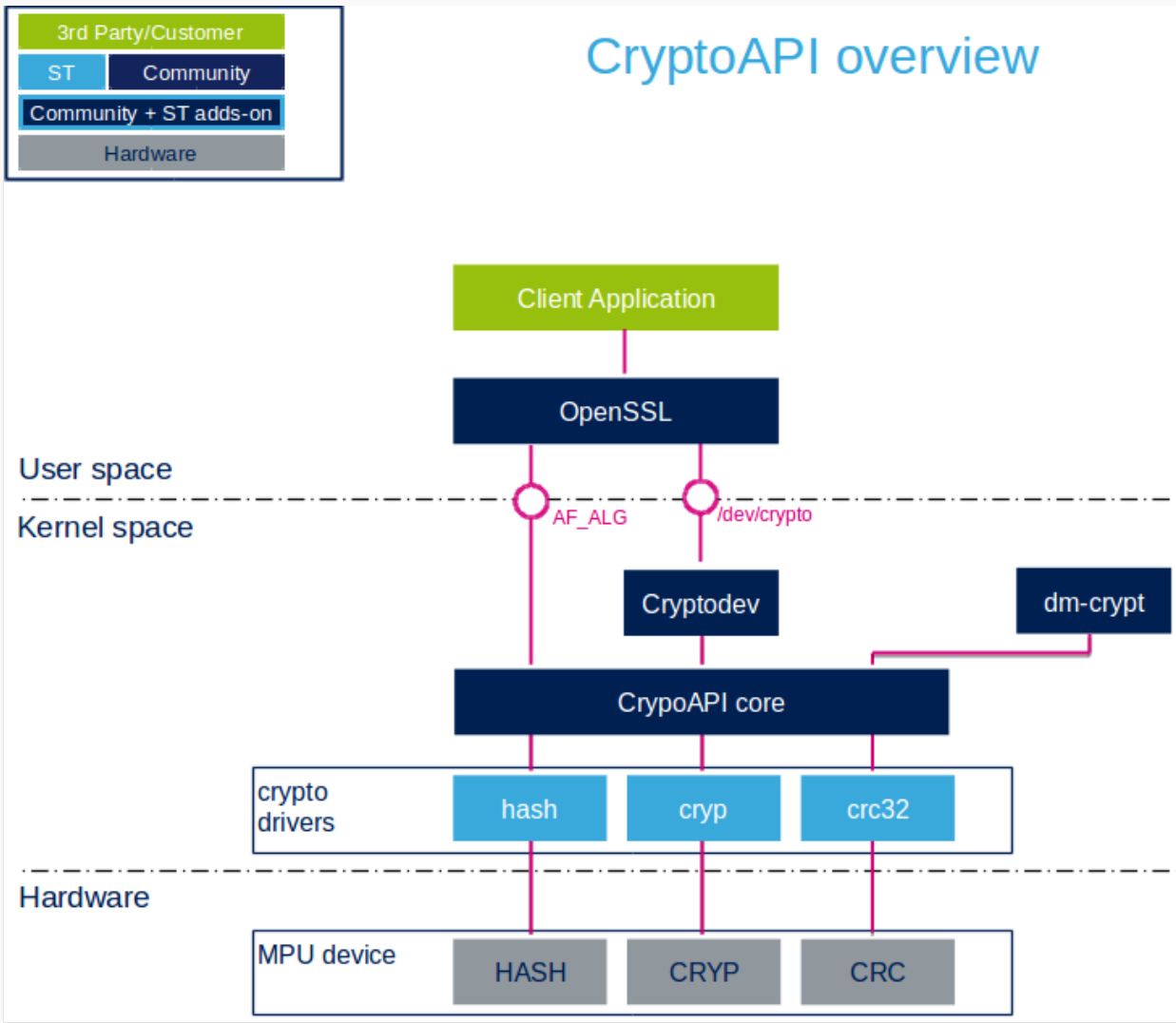
The Crypto API framework mainly includes all popular **hash** and **block ciphers** (encryption) functions.

A **hash** is a string or number generated from a text string. The length of the resulting string or number is fixed and widely varies with small variations of the input. The best hashing algorithms are designed so that it is impossible to turn a hash back into its original string. Hashing is particularly useful to compare a value with a stored value. However it cannot store its plain representation for security reasons. This makes hashing an ideal solution to store passwords.

Encryption turns data into a series of unreadable characters which length is not fixed. The encrypted strings can reversed back into their original decrypted form if the right key is not provided. Encrypting a confidential file is a good way to prevent anyone from accessing its content.

Drivers for CRYP (block cipher), HASH (hash) and CRC (cyclic redundancy check) are integrated within the Crypto API kernel service.

2 System overview



Crypto API

2.1 Description of the components

Information

OpenSSL and dm-crypt are not part of the Crypto API framework but they are typical users of the Crypto API services.

From User space to hardware

- **OpenSSL** (User space)

OpenSSL[1] is a software library supporting the TLS and SSL protocols as well as cryptographic functions. Openssl is available in OpenSTLinux distribution.

- **dm-crypt** (Kernel space)

dm-crypt[2] is a kernel disk encryption subsystem. It is natively available in the standard Linux kernel.

- **Cryptodev** (Kernel space)

Cryptodev[3] is a device driver which provides a general interface for userland applications. Although it is not part of the standard Linux kernel, it is available in OpenSTLinux distribution.

- **CryptoAPI core** (Kernel space)

This layer represents the standard Linux kernel cryptographic framework.

- **hash**, **cryp** and **crc32** (Kernel space)

These are the cryptographic Linux drivers handling the internal peripherals.

- **HASH**, **CRYP** and **CRC** (Hardware)

These HW blocks handle hash, ciphering, and CRC checksum.

2.2 API description

The Crypto API is documented in the Linux Kernel Crypto API section of the Linux Kernel documentation[4]. It offers both a kernel and a userland interface:

- kernel internal interface, used in particular by dm-crypt.
- userland algorithm interface (socket) named AF_ALG[5]. OpenSSL can use this interface.

In addition to the socket user interface, a more friendly interface, the cryptodev, can be used. It offers the /dev/crypto ioctl API. It is roughly described by the cryptodev.h[6] header file. OpenSSL can be configured to use this interface as an alternative to the historical AF_ALG interface.

3 Configuration

3.1 Kernal configuration

The Crypto API is activated by default in ST deliveries. Nevertheless, if a specific configuration is required, you can use Linux Menuconfig tool: Menuconfig or how to configure kernel and select:

```
[*] Cryptographic API  --->
    [*]   Hardware crypto devices  --->
        [*]   Support for STM32 crc accelerators
        [*]   Support for STM32 hash accelerators
        [*]   Support for STM32 crypto accelerators
```

3.2 Devicetree configuration

By default the drivers are not enabled, so this needs to be added if you want to use HW accelerators.

- crc: CRC_device_tree_configuration.
- hash: HASH_device_tree_configuration.
- crypto: CRYP_device_tree_configuration.

4 How to use the Crypto API framework

The Crypto API framework can be used by other kernel modules.

The Crypto API documentation provides kernel code examples[7]:

- Symmetric-key cipher operation.
- Operational state memory with SHASH.

5 Use cases

- Disk encryption

This is a typical example of Crypto API framework usage. Refer to LUKS[8] for a standard disk encryption process.

6 How to trace and debug the framework

6.1 How to monitor

The list of available ciphers is given in /proc/crypto:

```
Board $> cat /proc/crypto
```

Output part showing that an STM32 driver provides with the CRC32 cipher:

```
...
name         : crc32
driver       : stm32-crc32
module       : kernel
priority     : 200
refcnt       : 1
selftest     : passed
internal     : no
type         : shash
blocksize    : 1
digestsize   : 4
...
```

6.2 How to trace

There are no specific traces for this framework.

6.3 How to debug

There are no specific debug means for this framework.

7 Generic source code location

- CryptoAPI core
- CryptoAPI interface
- stm32 crypto drivers

8 References

- OpenSSL a software library supporting the TLS and SSL protocols as well as cryptographic functions.
- dm-crypt a kernel disk encryption subsystem
- Cryptodev a device driver which provides a general interface for userland applications
- Linux Kernel Crypto API the official crypto API kernel documentation
- Crypto API Userland interface specification of the userland API
- cryptodev.h cryptodev header file specifying the userland API
- Crypto API kernel code examples some kernel code examples using the Crypto API framework
- LUKS (Linux Unified Key Setup) a disk encryption specification

Application programming interface

Linux® is a registered trademark of Linus Torvalds.

Cryptographic processor

Cyclic redundancy check calculation unit

GPIO alternate function

Stable: 13.05.2020 - 08:37 / Revision: 13.05.2020 - 08:36

A quality version of this page, approved on *13 May 2020*, was based off this revision.

Contents

1 Article purpose 24
2 DT bindings documentation 25
3 DT configuration 26
3.1 DT configuration (STM32 level) 26
3.2 DT configuration (board level) 26
3.3 DT configuration examples 26
4 How to configure the DT using STM32CubeMX 27
5 References 28

1 Article purpose

This article explains how to configure the **HASH** internal peripheral when it is assigned to the Linux®OS. In that case, it is controlled by the Crypto framework.

The configuration is performed using the device tree mechanism that provides a hardware description of the HASH peripheral, used by the STM32 HASH Linux driver.

If the peripheral is assigned to another execution context, refer to How to assign an internal peripheral to a runtime context article for guidelines on peripheral assignment and configuration.

2 DT bindings documentation

The HASH is represented by the STM32 HASH device tree bindings[1]

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 HASH node is declared in stm32mp151.dtsi[2]. It describes the hardware register address, clock, interrupt, reset and dma.

```
 hash1: hash@54002000 {                                                    Comments
         compatible = "st,stm32f756-hash";
         reg = <0x54002000 0x400>;                                          -->
Register location and length
         interrupts = <GIC_SPI 80 IRQ_TYPE_LEVEL_HIGH>;                     --> The
interrupt number used
         clocks = <&scmi0_clk CK_SCMI0_HASH1>;
         resets = <&scmi0_reset RST_SCMI0_HASH1>;
         dmas = <&mdma1 31 0x10 0x1000A02 0x0 0x0 0x0>;                     --> DMA
specifiers[3]
         dma-names = "in";
         dma-maxburst = <2>;
         status = "disabled";
 };
```

Warning

This device tree part is related to STM32 microprocessors. It must be kept as is, without being modified by the end-user.

3.2 DT configuration (board level)

This part is used to enable the HASH used on a board which is done by setting the **status** property to **okay**.

3.3 DT configuration examples

```
&hash1 {
        status = "okay";
};
```

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.
The STM32CubeMX may not support all the properties described in the above DT bindings documentation paragraph. If so, the tool inserts **user sections** in the generated device tree. These sections can then be edited to add some properties and they are preserved from one generation to another. Refer to STM32CubeMX user manual for further information.

5 References

Please refer to the following links for additional information:

- Device tree bindings
- STM32MP151 device tree file
- Documentation/devicetree/bindings/dma/stm32-mdma.txt , STM32 MDMA controller

Linux® is a registered trademark of Linus Torvalds.

Operating System

Device Tree

Generic Interrupt Controller

Serial Peripheral Interface

Direct Memory Access