Hi3861 V100 / Hi3861L V100 eFUSE

User Guide

Issue	**01**
Date	**2020-04-30**

HiSilicon (Shanghai) Technologies Co., Ltd.

Address:	New R&D Center, 49 Wuhe Road, Bantian, Longgang District, Shenzhen 518129 P. R. China
Website:	https://www.hisilicon.com/en/
Email:	support@hisilicon.com

About This Document

Purpose

This document describes the structure of the eFUSE controller of Hi3861 V100 and Hi3861L V100 and the API usage.

Related Versions

The following table lists the product versions related to this document.

Product Name	Version
Hi3861	V100
Hi3861L	V100

Intended Audience

This document is intended for:

- Technical support engineers
- Software development engineers

Symbol Conventions

The following table describes the symbols that may be found in this document.

Symbol	Description
⚠ DANGER	Indicates a hazard with a high level of risk which, if not avoided, will result in death or serious injury.
⚠ WARNING	Indicates a hazard with a medium level of risk which, if not avoided, could result in death or serious injury.

Symbol	Description
⚠ CAUTION	Indicates a hazard with a low level of risk which, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. NOTICE is used to address practices not related to personal injury.
📖 NOTE	Supplements the important information in the main text. NOTE is used to address information not related to personal injury, equipment damage, and environment deterioration.

Change History

Issue	Date	Change Description
01	2020-04-30	This issue is the first official release. ● **Table 2-1** is updated.
00B02	2020-02-12	● **Table 2-1** is updated. ● **Table 3-1** is updated. ● Enumerated eFUSE parameter field IDs in **3.1 API Description** are updated.
00B01	2020-01-15	This issue is the first draft release.

Contents

1 Introduction

The eFUSE is a programmable storage unit. Because it can be programmed only once, it is mainly used to store the chip ID, key, or other one-time data. Hi3861 provides a 2 KB eFUSE space (bit[2047:0]). The eFUSE space is divided into multiple parameter fields based on the usage. You can program each bit of each field and lock the programming of each field using a lock bit with the control of the software.

NOTICE

As a one-time programmable storage unit, the eFUSE bit cannot be restored to **0** once it is burnt to **1**.

NOTE

For details about the eFUSE registers mentioned in this document, see the *Hi3861 V100/ Hi3861L V100/Hi3881 V100 Wi-Fi Chip Data Sheet*.

2 Function Description

2.1 Structure

The eFUSE consists of 94 fields, including:

- 51 system parameter fields
- 2 customer parameter fields
- 41 lock bits

The bit width of each parameter field is different. **Table 2-1** describes the attributes such as the field name and bit width of each parameter.

Table 2-1 Description of eFUSE parameter fields

Parameter	Bit Field	Bit Width	Lock Bit	Readable by Software	Writable by Software	Description
chip_id	bit[7:0]	8	PG0	1	0	Chip ID, including the chip model and flash vendor
die_id	bit[199:8]	192	PG1	1	0	Die ID, including the manufacturer and production date

Parameter	Bit Field	Bit Width	Lock Bit	Readable by Software	Writable by Software	Description
pmu_fuse1	bit[209:200]	10	PG2	1	0	For internal use
pmu_fuse2	bit[219:210]	10	PG2	1	0	For internal use
flash_encpt_cnt[7:6]	bit[221:220]	2	PG36	1	1	Firmware encryption/ decryption count 4, used for flash encryption, and automatically updated by HiBurn
flash_encpt_cnt[9:8]	bit[223:222]	2	PG37	1	1	Firmware encryption/ decryption count 5, used for flash encryption, and automatically updated by HiBurn
flash_encpt_cnt[11:10]	bit[225:224]	2	PG38	1	1	Firmware encryption/ decryption count 6, used for flash encryption, and automatically updated by HiBurn
secure_boot	bit[233:226]	8	PG39	1	1	Secure boot flag. 0x42 indicates non-secure boot, and other values indicate secure boot.
deep_sleep_flag	bit[234]	1	PG40	1	1	Deep sleep flag upon a verification failure. After **1** is written, if the FlashBoot verification has failed for six times, the chip enters the deep sleep state. By default, the chip is restarted due to verification failure.
PG36	bit[235]	1	-	1	1	PG36. Setting this bit to **1** locks the **flash_encpt_cnt[7:6]** field.

Parameter	Bit Field	Bit Width	Lock Bit	Readable by Software	Writable by Software	Description
PG37	bit[236]	1	-	1	1	PG37. Setting this bit to **1** locks the **flash_encpt_cnt[9:8]** field.
PG38	bit[237]	1	-	1	1	PG38. Setting this bit to **1** locks the **flash_encpt_cnt[11:10]** field.
PG39	bit[238]	1	-	1	1	PG39. Setting this bit to **1** locks the **secure_boot** field.
PG40	bit[239]	1	-	1	1	PG40. Setting this bit to **1** locks the **deep_sleep_flag** field.
root_pubkey	bit[495:240]	256	PG3	1	1	Hash value of the root public key, which is used for secure boot
root_key	bit[751:496]	256	PG4	0	1	Root key, which is used for firmware encryption or HUKS
customer_rsvd0	bit[1007:752]	256	PG5	1	1	Reserved for the user
subkey_cat	bit[1039:1008]	32	PG6	1	1	Level-2 key, which is used for secure boot
encrypt_flag	bit[1047:1040]	8	PG7	1	1	Encryption flag of the flashboot and loaderboot code segment. The value **0x42** indicates that the code segment is not encrypted, and other values indicate that the code segment is encrypted. The flag is used for secure boot.

Parameter	Bit Field	Bit Width	Lock Bit	Readable by Software	Writable by Software	Description
rsim	bit[1071:1048]	24	PG7	1	1	Level-2 key revocation identifier, which is used for secure boot
start_type	bit[1072]	1	PG2	1	0	For internal use
jtm	bit[1073]	1	PG8	1	1	JTAG mask. Setting this bit to **1** masks the JTAG function.
utm0	bit[1074]	1	PG9	1	1	UART0 mask. Setting this bit to **1** masks the UART0 function.
utm1	bit[1075]	1	PG10	1	1	UART1 mask. Setting this bit to **1** masks the UART1 function.
utm2	bit[1076]	1	PG11	1	1	UART2 mask. Setting this bit to **1** masks the UART2 function.
sdc	bit[1077]	1	PG12	1	0	For internal use
rsvd0	bit[1078]	1	PG13	1	1	Reserved
kdf2ecc_huk_disable	bit[1079]	1	PG35	1	0	For internal use
SSS_corner	bit[1081:1080]	2	PG14	1	0	For internal use

Parameter	Bit Field	Bit Width	Lock Bit	Readable by Software	Writable by Software	Description
uart_halt_interval	bit[1083:1082]	2	PG15	1	1	Interval time for waiting for HiBurn connection during boot. The shorter the time, the faster the boot. The values 0–3 correspond to 32 ms, 1 ms, 2 ms, and 16 ms respectively. If the interval is shortened to 1 ms or 2 ms, the interval for sending packets on HiBurn must be changed to 2 ms, which reduces the HiBurn connection success rate.
ts_trim	bit[1087:1084]	4	PG2	1	0	For internal use
chip_id2	bit[1095:1088]	8	PG16	1	0	Chip ID backup
ipv4_mac_addr	bit[1143:1096]	48	PG17	1	1	IPv4 MAC address
ipv6_mac_addr	bit[1271:1144]	128	PG17	1	1	IPv6 MAC address
pa2gccka0_trim0	bit[1303:1272]	32	PG18	1	1	802.11b core 0, power trimming factor, round 1
pa2gccka1_trim0	bit[1335:1304]	32	PG18	1	1	802.11b core 0, power trimming factor, round 1

Parameter	Bit Field	Bit Width	Lock Bit	Readable by Software	Writable by Software	Description
nvram_pa2ga0_trim0	bit[1367:1336]	32	PG19	1	1	802.11g 20 MHz, power trimming factor, round 1
nvram_pa2ga1_trim0	bit[1399:1368]	32	PG19	1	1	802.11g 20 MHz, power trimming factor, round 1
pa2gccka0_trim1	bit[1431:1400]	32	PG20	1	1	802.11b core 0, power trimming factor, round 2
pa2gccka1_trim1	bit[1463:1432]	32	PG20	1	1	802.11b core 0, power trimming factor, round 2
nvram_pa2ga0_trim1	bit[1495:1464]	32	PG21	1	1	802.11g 20 MHz, power trimming factor, round 2
nvram_pa2ga1_trim1	bit[1527:1496]	32	PG21	1	1	802.11g 20 MHz, power trimming factor, round 2
pa2gccka0_trim2	bit[1559:1528]	32	PG22	1	1	802.11b core 0, power trimming factor, round 3
pa2gccka1_trim2	bit[1591:1560]	32	PG22	1	1	802.11b core 0, power trimming factor, round 3
nvram_pa2ga0_trim2	bit[1623:1592]	32	PG23	1	1	802.11g 20 MHz, power trimming factor, round 3
nvram_pa2ga1_trim2	bit[1655:1624]	32	PG23	1	1	802.11g 20 MHz, power trimming factor, round 3

Parameter	Bit Field	Bit Width	Lock Bit	Readable by Software	Writable by Software	Description
tee_boot_ver	bit[1671:1656]	16	PG24	1	1*	TEE boot version, which is updated during program burning or upgrade in secure boot mode
tee_firmware_ver	bit[1719:1672]	48	PG25	1	1*	TEE firmware version, which is updated during program burning or upgrade in secure boot mode
tee_salt	bit[1847:1720]	128	PG26	1*	1*	TEE salt value, which is automatically written by the HUKS software
flash_encpt_cnt[1:0]	bit[1849:1848]	2	PG27	1	1	Firmware encryption/ decryption count 1, used for flash encryption, and automatically updated by HiBurn
flash_encpt_cnt[3:2]	bit[1851:1850]	2	PG28	1	1	Firmware encryption/ decryption count 2, used for flash encryption, and automatically updated by HiBurn
flash_encpt_cnt[5:4]	bit[1853:1852]	2	PG29	1	1	Firmware encryption/ decryption count 3, used for flash encryption, and automatically updated by HiBurn
flash_encpt_cfg	bit[1854]	1	PG30	1	1	Firmware encryption/ decryption control bit. When this bit is set to **1**, the flash encryption function is enabled.
flash_scramble_en	bit[1855]	1	PG31	1	1	Flash data scrambling enable
user_flash_ind	bit[1865:1856]	10	PG31	1	1	Salt for flash data scrambling

Parameter	Bit Field	Bit Width	Lock Bit	Readable by Software	Writable by Software	Description
rf_pdbuffer_gcal	bit[1883:1866]	18	PG32	1	0	For internal use
customer_rsvd1	bit[1947:1884]	64	PG33	1	1	Reserved for the user
die_id2	bit[2011:1948]	64	PG34	1	0	Die ID 2
PG0	bit[2012]	1	-	1	0	PG0. Setting this bit to **1** locks the **chip_id** field.
PG1	bit[2013]	1	-	1	0	PG1. Setting this bit to **1** locks the **die_id** field.
PG2	bit[2014]	1	-	1	0	PG2. Setting this bit to **1** locks the **pmu_fuse1**, **pmu_fuse2**, **start_type**, and **ts_trim** fields.
PG3	bit[2015]	1	-	1	1	PG3. Setting this bit to **1** locks the **root_pubkey** field.
PG4	bit[2016]	1	-	1	1	PG4. Setting this bit to **1** locks the **root_key** field.
PG5	bit[2017]	1	-	1	1	PG5. Setting this bit to **1** locks the **customer_rsvd0** field.
PG6	bit[2018]	1	-	1	1	PG6. Setting this bit to **1** locks the **subkey_cat** field.
PG7	bit[2019]	1	-	1	1	PG7. Setting this bit to **1** locks the **encrypt_flag** and **rsim** fields.
PG8	bit[2020]	1	-	1	1	PG8. Setting this bit to **1** locks the **jtm** field.
PG9	bit[2021]	1	-	1	1	PG9. Setting this bit to **1** locks the **utm0** field.

Parameter	Bit Field	Bit Width	Lock Bit	Readable by Software	Writable by Software	Description
PG10	bit[2022]	1	-	1	1	PG10. Setting this bit to **1** locks the **utm1** field.
PG11	bit[2023]	1	-	1	1	PG11. Setting this bit to **1** locks the **utm2** field.
PG12	bit[2024]	1	-	1	0	PG12. Setting this bit to **1** locks the **sdc** field.
PG13	bit[2025]	1	-	1	1	PG13. Setting this bit to **1** locks the **rsvd0** field.
PG14	bit[2026]	1	-	1	0	PG14. Setting this bit to **1** locks the **SSS_corner** field.
PG15	bit[2027]	1	-	1	1	PG15. Setting this bit to **1** locks the **uart_halt_interval** field.
PG16	bit[2028]	1	-	1	0	PG16. Setting this bit to **1** locks the **chip_id2** field.
PG17	bit[2029]	1	-	1	1	PG17. Setting this bit to **1** locks the **ipv4_mac_addr** and **ipv6_mac_addr** fields.
PG18	bit[2030]	1	-	1	1	PG18. Setting this bit to **1** locks the **pa2gccka0_trim0** and **pa2gccka1_trim0** fields.
PG19	bit[2031]	1	-	1	1	PG19. Setting this bit to **1** locks the **nvram_pa2ga0_trim0** and **nvram_pa2ga1_trim0** fields.
PG20	bit[2032]	1	-	1	1	PG20. Setting this bit to **1** locks the **pa2gccka0_trim1** and **pa2gccka1_trim1** fields.

Parameter	Bit Field	Bit Width	Lock Bit	Readable by Software	Writable by Software	Description
PG21	bit[2033]	1	-	1	1	PG21. Setting this bit to **1** locks the **nvram_pa2ga0_trim1** and **nvram_pa2ga1_trim1** fields.
PG22	bit[2034]	1	-	1	1	PG22. Setting this bit to **1** locks the **pa2gccka0_trim2** and **pa2gccka2_trim2** fields.
PG23	bit[2035]	1	-	1	1	PG23. Setting this bit to **1** locks the **nvram_pa2ga0_trim2** and **nvram_pa2ga1_trim2** fields.
PG24	bit[2036]	1	-	1	1*	PG24. Setting this bit to **1** locks the **tee_boot_ver** field.
PG25	bit[2037]	1	-	1	1*	PG25. Setting this bit to **1** locks the **tee_firmware_ver** field.
PG26	bit[2038]	1	-	1	1*	PG26. Setting this bit to **1** locks the **tee_salt** field.
PG27	bit[2039]	1	-	1	1	PG27. Setting this bit to **1** locks the **flash_encpt_cnt[1:0]** field.
PG28	bit[2040]	1	-	1	1	PG28. Setting this bit to **1** locks the **flash_encpt_cnt[3:2]** field.
PG29	bit[2041]	1	-	1	1	PG29. Setting this bit to **1** locks the **flash_encpt_cnt[5:4]** field.
PG30	bit[2042]	1	-	1	1	PG30. Setting this bit to **1** locks the **flash_encpt_cfg** field.

Parameter	Bit Field	Bit Width	Lock Bit	Readable by Software	Writable by Software	Description
PG31	bit[2043]	1	-	1	1	PG31. Setting this bit to **1** locks the **flash_scramble_en** and **user_flash_ind** fields.
PG32	bit[2044]	1	-	1	0	PG32. Setting this bit to **1** locks the **rf_pdbuffer_gcal** field.
PG33	bit[2045]	1	-	1	1	PG33. Setting this bit to **1** locks the **customer_rsvd1** field.
PG34	bit[2046]	1	-	1	0	PG34. Setting this bit to **1** locks the **die_id2** field.
PG35	bit[2047]	1	-	1	0	PG35. Setting this bit to **1** locks the **kdf2ecc_huk_disable** field.

2.2 Burning

The software burns the eFUSE bit[2047:0] by controlling the eFUSE register. Before burning, all data in the eFUSE is **0**. After burning, the bit unit corresponding to the programming address is changed to **1**.

The **Writable by Software** attribute in **Table 2-1** describes whether the eFUSE fields can be burnt by software. The values are as follows:

- **0**: The fields cannot be burnt by software.
- **1**: The fields can be burnt by software.
- **1***: The corresponding eFUSE field can be burnt by software only when the **nmi_int_flag** interrupt is triggered. Otherwise, the addresses of bit[1847:1656] and bit[2038:2036] cannot be programmed by software. If the address programming is started when the software cannot burn the fields, the controller does not perform programming and an error is reported.

The software burning process is as follows:

Step 1 Read bit[4] of **EFUSE_CTRL_ST** to check whether the eFUSE is idle. If yes (read as **0**), read bit[2] to check whether the eFUSE is powered on. If yes (read as **1**), go to Step 2. Otherwise, wait until the eFUSE is powered on.

Step 2 Write bit[10:0] of **EFUSE_PGM_A** to set the address to be burnt.

Step 3 Write **1** to bit[0] of **EFUSE_PGM_EN** to enable the programming mode.

Step 4 Read bit[0] of **EFUSE_PGM_EN** to check whether the programming operation is complete. If yes (read as **0**), go to Step 3. Otherwise, wait until the programming operation is complete.

Step 5 Read bit[5] of **EFUSE_CTRL_ST** to check whether the burning is successful. If **EFUSE_CTRL_ST** is read as **1**, the burning fails. If **EFUSE_CTRL_ST** is read as **0** and bit[0] is read as **1**, the burning is successful.

----End

NOTICE

After the preceding steps are complete, check the current burning address bit[10:0] of **EFUSE_PGM_A**. If the value is **0x73E** or **0x73F**, the current unlocked parameter field is **flash_scramble_en** or **user_flash_ind**, and the burning takes effect immediately. If the value is another unlocked parameter field, power off the eFUSE and then power it on again for the burning to take effect.

2.3 Reading

The **Readable by Software** attribute in **Table 2-2** indicates whether the eFUSE fields can be read by software. The values are as follows:

- **0**: The fields cannot be read by software.
- **1**: The fields can be read by software.
- **1***: The corresponding eFUSE field can be read by software only when the **nmi_int_flag** interrupt is triggered. Otherwise, the eFUSE field cannot be read by software.

NOTICE

If the eFUSE parameter field cannot be read, attempts to read an unreadable area return all zeros.

The software reads data as follows:

Step 1 Read bit[4] of **EFUSE_CTRL_ST** to check whether the eFUSE is idle. If yes (read as **0**), read bit[2] to check whether the eFUSE is powered on. If yes (read as **1**), go to Step 2. Otherwise, wait until the eFUSE is powered on.

Step 2 Assign a value to bit[7:0] of **EFUSE_RD_A**. Each value corresponds to 8-bit data of the eFUSE. For details about the mapping, see **Table 2-2**.

Table 2-2 Mapping between read addresses and eFUSE data

EFUSE_RD_A bit[7:0]	0x0	0x1	0x2	......	0xFF
eFUSE Data	bit[7:0]	bit[15:8]	bit[23:16]	......	bit[2047:2040]

Step 3 Write **1** to bit[0] of **EFUSE_RD_EN** to enter the data read mode and read 8-bit data each time.

Step 4 Read bit[0] of **EFUSE_RD_EN** to check whether the read mode is complete. If **EFUSE_RD_EN** is read as **0**, the read mode is complete. If bit[1] of **EFUSE_CTRL_ST** is read as **1** at the same time, the read is complete.

Step 5 Read bit[7:0] of **EFUSE_RDATA** to obtain the required 8-bit data.

----End

2.4 Burning Locking

The software locks a corresponding unlocked parameter field by setting the lock bit to **1**. For details about the mapping, see the Lock Bit column in **Table 2-1**. The locked eFUSE field is not programmable by software. Attempts to program the locked address have no effect and an error is reported. For example, the lock bit corresponding to **flash_encpt_cnt[7:6]** is **PG36**. If the lock bit of **PG36** is **1**, **flash_encpt_cnt[7:6]** cannot be burnt. If a bit of **flash_encpt_cnt[7:6]** is **0** after being locked, the value of this bit is always **0** and cannot be changed.

NOTICE

The lock bit and unlock parameter field do not have a one-to-one mapping relationship. Some lock bits can lock multiple unlocked parameter fields, as shown in **Table 2-1**.

The process of burning locking by software is as follows:

Step 1 Read bit[4] of **EFUSE_CTRL_ST** to check whether the eFUSE is idle. If yes (read as **0**), read bit[2] to check whether the eFUSE is powered on. If yes (read as **1**), go to Step 2. Otherwise, wait until the eFUSE is powered on.

Step 2 Write bit[10:0] of **EFUSE_PGM_A** to set the corresponding address. The address range of the lock bit is **0xEB–0xEF** and **0x7F0–0x7FF**.

Step 3 Write **1** to bit[0] of **EFUSE_PGM_EN** to enable the programming mode.

Step 4 Read bit[0] of **EFUSE_PGM_EN** to check whether the programming operation is complete. If yes (read as **0**), go to Step 3. Otherwise, wait until the programming operation is complete.

Step 5 Read bit[5] of **EFUSE_CTRL_ST** to check whether the burning by software is successful. If **EFUSE_CTRL_ST** is read as **1**, the burning fails. If **EFUSE_CTRL_ST** is read as **0** and bit[0] is read as **1**, the burning is successful.

----End

NOTICE

After the preceding steps are complete, check the current burning address bit[10:0] of **EFUSE_PGM_A**. If the value is **0x7FB**, the current burning lock bit is **PG31**, and the lock takes effect immediately. If the burning lock bit is not **PG31**, you need to power off the eFUSE and then power it on again for the burning locking to take effect.

3 API Usage Guide

3.1 API Description

Based on the preceding process, the eFUSE module provides two sets of software APIs:

- ID indexing mode: assigns IDs to the unlocked parameter fields and locked parameter fields respectively. (For the mapping between the software unlocked parameter field ID, locked parameter field ID, and eFUSE field, see **Table 3-1**.) The APIs are implemented by indexing the corresponding eFUSE field based on the ID. The following APIs are provided:
  - **hi_efuse_get_id_size**: Obtains the length (in bits) of an eFUSE field by field ID.
  - **hi_efuse_read**: Reads data from an eFUSE field by field ID.
  - **hi_efuse_write**: Writes data to an eFUSE field by field ID.
  - **hi_efuse_lock**: Locks an area in the eFUSE by using the lock ID. After the area is locked, data cannot no longer be written to it.
  - **hi_efuse_get_lockstat**: Obtains the lock status of the eFUSE and queries which areas are locked.
- Start address mode: specifies the read/write mode of the start address of the eFUSE. This mode is mainly used for the user reserved area. The following APIs are provided:
  - **hi_efuse_usr_read**: Reads the eFUSE from a specified start address.
  - **hi_efuse_usr_write**: Writes the eFUSE into a specified start address.

Table 3-1 Mapping between eFUSE parameters and bit fields

Parameter	Unlocked Parameter Field ID	Lock Bit ID
chip_id	0	-
die_id	1	-
pmu_fuse1	2	-
pmu_fuse2	3	-
flash_encpt_cnt[7:6]	4	-
flash_encpt_cnt[9:8]	5	-
flash_encpt_cnt[11:10]	6	-
secure_boot	52	-
deep_sleep_flag	7	-
PG36	-	36
PG37	-	37
PG38	-	38
PG39	-	39
PG40	-	40
root_pubkey	8	-
root_key	9	-
rsvd0	10	-
subkey_cat	11	-
encrypt_flag	12	-
rsim	13	-
start_type	14	-
jtm	15	-
utm0	16	-
utm1	17	-
utm2	18	-
sdc	19	-
rsvd1	20	-
kdf2ecc_huk_disable	21	-
SSS_corner	22	-

Parameter	Unlocked Parameter Field ID	Lock Bit ID
uart_halt_interval	23	-
ts_trim	24	-
chip_id2	25	-
ipv4_mac_addr	26	-
ipv6_mac_addr	27	-
pa2gccka0_trim0	28	-
pa2gccka1_trim0	29	-
nvram_pa2ga0_trim0	30	-
nvram_pa2ga1_trim0	31	-
pa2gccka0_trim1	32	-
pa2gccka1_trim1	33	-
nvram_pa2ga0_trim1	34	-
nvram_pa2ga1_trim1	35	-
pa2gccka0_trim2	36	-
pa2gccka1_trim2	37	-
nvram_pa2ga0_trim2	38	-
nvram_pa2ga1_trim2	39	-
tee_boot_ver	40	-
tee_firmware_ver	41	-
tee_salt	42	-
flash_encpt_cnt[1:0]	43	-
flash_encpt_cnt[3:2]	44	-
flash_encpt_cnt[5:4]	45	-
flash_encpt_cfg	46	-
flash_scramble_en	47	-
user_flash_ind	48	-
rf_pdbuffer_gcal	49	-
customer_rsvd0	50	-
customer_rsvd1	51	-

Parameter	Unlocked Parameter Field ID	Lock Bit ID
PG0	-	0
PG1	-	1
PG2	-	2
PG3	-	3
PG4	-	4
PG5	-	5
PG6	-	6
PG7	-	7
PG8	-	8
PG9	-	9
PG10	-	10
PG11	-	11
PG12	-	12
PG13	-	13
PG14	-	14
PG15	-	15
PG16	-	16
PG17	-	17
PG18	-	18
PG19	-	19
PG20	-	20
PG21	-	21
PG22	-	22
PG23	-	23
PG24	-	24
PG25	-	25
PG26	-	26
PG27	-	27
PG28	-	28

Parameter	Unlocked Parameter Field ID	Lock Bit ID
PG29	-	29
PG30	-	30
PG31	-	31
PG32	-	32
PG33	-	33
PG34	-	34
PG35	-	35

Enumerated eFUSE parameter field IDs:

```
typedef enum {
HI_EFUSE_CHIP_RW_ID = 0,
HI_EFUSE_DIE_RW_ID = 1,
HI_EFUSE_PMU_FUSE1_RW_ID = 2,
HI_EFUSE_PMU_FUSE2_RW_ID = 3,
HI_EFUSE_FLASH_ENCPY_CNT3_RW_ID = 4,
HI_EFUSE_FLASH_ENCPY_CNT4_RW_ID = 5,
HI_EFUSE_FLASH_ENCPY_CNT5_RW_ID = 6,
HI_EFUSE_DSLEEP_FLAG_RW_ID = 7,
HI_EFUSE_ROOT_PUBKEY_RW_ID = 8,
HI_EFUSE_ROOT_KEY_WO_ID = 9,
HI_EFUSE_CUSTOMER_RSVD0_RW_ID = 10,
HI_EFUSE_SUBKEY_CAT_RW_ID = 11,
HI_EFUSE_ENCRYPT_FLAG_RW_ID = 12,
HI_EFUSE_SUBKEY_RSIM_RW_ID = 13,
HI_EFUSE_START_TYPE_RW_ID = 14,
HI_EFUSE_JTM_RW_ID = 15,
HI_EFUSE_UTM0_RW_ID = 16,
HI_EFUSE_UTM1_RW_ID = 17,
HI_EFUSE_UTM2_RW_ID = 18,
HI_EFUSE_SDC_RW_ID = 19,
HI_EFUSE_RSVD0_RW_ID = 20,
HI_EFUSE_KDF2ECC_HUK_DISABLE_RW_ID = 21,
HI_EFUSE_SSS_CORNER_RW_ID = 22,
HI_EFUSE_UART_HALT_INTERVAL_RW_ID = 23,
HI_EFUSE_TSENSOR_RIM_RW_ID = 24,
HI_EFUSE_CHIP_BK_RW_ID = 25,
HI_EFUSE_IPV4_MAC_ADDR_RW_ID = 26,
HI_EFUSE_IPV6_MAC_ADDR_RW_ID = 27,
HI_EFUSE_PG2GCCKA0_TRIM0_RW_ID = 28,
HI_EFUSE_PG2GCCKA1_TRIM0_RW_ID = 29,
HI_EFUSE_NVRAM_PA2GA0_TRIM0_RW_ID = 30,
HI_EFUSE_NVRAM_PA2GA1_TRIM0_RW_ID = 31,
HI_EFUSE_PG2GCCKA0_TRIM1_RW_ID = 32,
HI_EFUSE_PG2GCCKA1_TRIM1_RW_ID = 33,
HI_EFUSE_NVRAM_PA2GA0_TRIM1_RW_ID = 34,
HI_EFUSE_NVRAM_PA2GA1_TRIM1_RW_ID = 35,
HI_EFUSE_PG2GCCKA0_TRIM2_RW_ID = 36,
HI_EFUSE_PG2GCCKA1_TRIM2_RW_ID = 37,
HI_EFUSE_NVRAM_PA2GA0_TRIM2_RW_ID = 38,
HI_EFUSE_NVRAM_PA2GA1_TRIM2_RW_ID = 39,
HI_EFUSE_TEE_BOOT_VER_RW_ID = 40,
HI_EFUSE_TEE_FIRMWARE_VER_RW_ID = 41,
HI_EFUSE_TEE_SALT_RW_ID = 42,
HI_EFUSE_FLASH_ENCPY_CNT0_RW_ID = 43,
HI_EFUSE_FLASH_ENCPY_CNT1_RW_ID = 44,
HI_EFUSE_FLASH_ENCPY_CNT2_RW_ID = 45,
HI_EFUSE_FLASH_ENCPY_CFG_RW_ID = 46,
HI_EFUSE_FLASH_SCRAMBLE_EN_RW_ID = 47,
HI_EFUSE_USER_FLASH_IND_RW_ID = 48,
HI_EFUSE_RF_PDBUFFER_GCAL_RW_ID = 49,
HI_EFUSE_CUSTOMER_RSVD1_RW_ID = 50,
HI_EFUSE_DIE_2_RW_ID = 51,
HI_EFUSE_SEC_BOOT_RW_ID = 52,
HI_EFUSE_IDX_MAX,
} hi_efuse_idx;
```

Enumerated lock bit IDs:

```
typedef enum {
HI_EFUSE_LOCK_CHIP_ID = 0,
HI_EFUSE_LOCK_DIE_ID = 1,
HI_EFUSE_LOCK_PMU_FUSE1_FUSE2_START_TYPE_TSENSOR_ID = 2,
HI_EFUSE_LOCK_ROOT_PUBKEY_ID = 3,
HI_EFUSE_LOCK_ROOT_KEY_ID = 4,
HI_EFUSE_LOCK_CUSTOMER_RSVD0_ID = 5,
HI_EFUSE_LOCK_SUBKEY_CAT_ID = 6,
HI_EFUSE_LOCK_ENCRYPT_RSIM_ID = 7,
HI_EFUSE_LOCK_JTM_ID = 8,
HI_EFUSE_LOCK_UTM0_ID = 9,
HI_EFUSE_LOCK_UTM1_ID = 10,
HI_EFUSE_LOCK_UTM2_ID = 11,
HI_EFUSE_LOCK_SDC_ID = 12,
HI_EFUSE_LOCK_RSVD0_ID = 13,
HI_EFUSE_LOCK_SSS_CORNER_ID = 14,
HI_EFUSE_LOCK_UART_HALT_INTERVAL_ID = 15,
HI_EFUSE_LOCK_CHIP_BK_ID = 16,
HI_EFUSE_LOCK_IPV4_IPV6_MAC_ADDR_ID = 17,
HI_EFUSE_LOCK_PG2GCCKA0_PG2GCCKA1_TRIM0_ID = 18,
HI_EFUSE_LOCK_NVRAM_PA2GA0_PA2GA1_TRIM0_ID = 19,
HI_EFUSE_LOCK_PG2GCCKA0_PG2GCCKA1_TRIM1_ID = 20,
HI_EFUSE_LOCK_NVRAM_PA2GA0_PA2GA1_TRIM1_ID = 21,
HI_EFUSE_LOCK_PG2GCCKA0_PG2GCCKA1_TRIM2_ID = 22,
HI_EFUSE_LOCK_NVRAM_PA2GA0_PA2GA1_TRIM2_ID = 23,
HI_EFUSE_LOCK_TEE_BOOT_VER_ID = 24,
HI_EFUSE_LOCK_TEE_FIRMWARE_VER_ID = 25,
HI_EFUSE_LOCK_TEE_SALT_ID = 26,
HI_EFUSE_LOCK_FLASH_ENCPY_CNT0_ID = 27,
HI_EFUSE_LOCK_FLASH_ENCPY_CNT1_ID = 28,
HI_EFUSE_LOCK_FLASH_ENCPY_CNT2_ID = 29,
HI_EFUSE_LOCK_FLASH_ENCPY_CFG_ID = 30,
HI_EFUSE_LOCK_FLASH_SCRAMBLE_EN_FLASH_IND_ID = 31,
HI_EFUSE_LOCK_RF_PDBUFFER_GCAL_ID = 32,
HI_EFUSE_LOCK_CUSTOMER_RSVD1_ID = 33,
HI_EFUSE_LOCK_DIE_2_ID = 34,
HI_EFUSE_LOCK_KDF2ECC_HUK_DISABLE_ID = 35,
HI_EFUSE_LOCK_FLASH_ENCPY_CNT3_ID = 36,
HI_EFUSE_LOCK_FLASH_ENCPY_CNT4_ID = 37,
HI_EFUSE_LOCK_FLASH_ENCPY_CNT5_ID = 38,
HI_EFUSE_LOCK_SEC_BOOT_ID = 39,
HI_EFUSE_LOCK_DSLEEP_FLAG_ID = 40,
HI_EFUSE_LOCK_MAX,
} hi_efuse_lock_id;
```

3.2 Development Procedure

The following uses the **customer_rsvd0** parameter field as an example to describe how to use the software APIs in ID indexing mode:

Step 1 Write data to the eFUSE by calling **hi_efuse_write**.

Step 2 Read data from the eFUSE by calling **hi_efuse_read**.

NOTICE

When reading the eFUSE, ensure that the space for storing the read data is greater than or equal to the length of the data to be read.

Step 3 Lock an area in the eFUSE by calling **hi_efuse_lock**. After the area is locked, data cannot be written to it.

----End

The following uses the **customer_rsvd0** parameter field as an example to describe how to use the software APIs in start address mode:

Step 1 Write data to the eFUSE by calling **hi_efuse_usr_write**.

Step 2 Read data from the eFUSE by calling **hi_efuse_usr_read**.

NOTICE

- Only 8-bit data can be read from the eFUSE at a time by directly reading the address, as shown in **Table 2-2**. Therefore, when the **hi_efuse_usr_read** API is called to read the eFUSE, the start address and read length need to be 8-bit aligned. The read data needs to be shifted accordingly. For example, if the input start address is 8-bit aligned and the number of bits to be read is 10, the data to be read can be obtained only after the read data is shifted rightwards by 6 bits.
- When reading the eFUSE, ensure that the space for storing the read data is greater than or equal to the length of the data to be read.

Step 3 Call **hi_efuse_usr_write** to set the burning lock bit to **1** to lock a certain area in the eFUSE. After the area is locked, data cannot be written to the area.

----End

3.3 Code Sample

Sample 1: The **customer_rsvd0** field is read, written, and locked by ID.

```
#define EFUSE_USR_RW_SAMPLE_BUFF_MAX_LEN 2 /* The length of customer_rsvd0 of the
eFUSE is 64 bits. Two 32-bit spaces are required for storing the read and write data. */
```

```
hi_void efuse_get_lock_stat(hi_void)
{
    hi_u64 lock_data;

    hi_efuse_get_lockstat(&lock_data);
    printf("lock_stat = 0x%08X ", (hi_u32)((lock_data >> 32) & 0xFFFFFFFF)); /* right shift 32bits
*/
    printf("%08X\n", (hi_u32)(lock_data & 0xFFFFFFFF));
}

hi_u32 efuse_id_read(hi_void)
{
    hi_u32 ret;
    hi_u32 read_data[EFUSE_USR_RW_SAMPLE_BUFF_MAX_LEN] = {0};
    hi_efuse_idx efuse_id = HI_EFUSE_CUSTOMER_RSVD0_RW_ID;

    ret = hi_efuse_read(efuse_id, (hi_u8 *)read_data, (hi_u8)sizeof(read_data));
    if (ret != HI_ERR_SUCCESS) {
        printf("Failed to read EFUSE at line%d! Err code = %X\n", __LINE__, ret);
        return ret;
    }
    printf("id_data = 0x%08X %08X\n", read_data[0], read_data[1]);

    return HI_ERR_SUCCESS;
}

hi_u32 efuse_id_write(hi_void)
{
    hi_u32 ret;
    hi_u32 write_data[EFUSE_USR_RW_SAMPLE_BUFF_MAX_LEN] = {
        0x1,
        0x0,
    };
    hi_efuse_idx efuse_id = HI_EFUSE_CUSTOMER_RSVD0_RW_ID;

    ret = efuse_id_read();
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }

    ret = hi_efuse_write(efuse_id, (hi_u8 *)write_data);
    if (ret != HI_ERR_SUCCESS) {
        printf("Failed to write EFUSE!\n");
        return ret;
    }

    return HI_ERR_SUCCESS;
}

hi_u32 efuse_id_lock(hi_void)
{
    hi_u32 ret;
    hi_efuse_lock_id lock_id = HI_EFUSE_LOCK_CUSTOMER_RSVD0_ID;
    efuse_get_lock_stat();

    ret = hi_efuse_lock(lock_id);
    if (ret != HI_ERR_SUCCESS) {
        printf("Failed to lock EFUSE!\n");
        return ret;
    }

    efuse_get_lock_stat();
    return HI_ERR_SUCCESS;
}

hi_u32 sample_id_efuse(hi_void)
{
    hi_u32 ret;

#ifdef EFUSE_WRITE_ENABLE
    ret = efuse_id_write();
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }
#endif

    ret = efuse_id_read();
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }

#ifdef EFUSE_LOCK_ENABLE
    ret = efuse_id_lock();
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }
#endif

    return HI_ERR_SUCCESS;
}
```

Sample 2: The **customer_rsvd0** field is read, written, and locked by start address.

```
#define EFUSE_USR_RW_SAMPLE_BUFF_MAX_LEN 2 /* The length of customer_rsvd0 of the
eFUSE is 64 bits. Two 32-bit spaces are required for storing the read and write data. */
hi_void efuse_get_lock_stat(hi_void)
{
    hi_u64 lock_data;

    hi_efuse_get_lockstat(&lock_data);
    printf("lock_stat = 0x%08X ", (hi_u32)((lock_data >> 32) & 0xFFFFFFFF)); /* right shift 32bits
*/
    printf("%08X\n", (hi_u32)(lock_data & 0xFFFFFFFF));
}

hi_u32 efuse_usr_read(hi_void)
{
    hi_u32 ret;
    hi_u32 read_data[EFUSE_USR_RW_SAMPLE_BUFF_MAX_LEN] = {0};
    hi_u16 start_bit = 0x75C;       /* The offset address of customer_rsvd0 is 0x75C. */
    hi_u16 rw_bits = 64;            /* The length of customer_rsvd0 is 64 bits. */
    hi_u16 align_size;
    hi_u8 diff_head_read = 0;
    hi_u8 tmp_data[9] = {0};        /* The 8-bit-aligned address and length of customer_rsvd0 is 9
bytes (72 bits). */
    hi_u64 first_u64;
    hi_u8 second_u8;

    if ((start_bit & 0x7) != 0x0) {
        diff_head_read = start_bit % 8; /* Read the 8-bit-aligned start address. */
        start_bit = start_bit - diff_head_read;
        align_size = rw_bits + diff_head_read;
    }
    if ((align_size & 0x7) != 0x0) {
        align_size = ((align_size >> 3) + 1) << 3; /* 3-bit offset: The read length is in the unit of 8 
bits. */
    }

    ret = hi_efuse_usr_read(start_bit, align_size, (hi_u8 *)tmp_data);
    if (ret != HI_ERR_SUCCESS) {
        printf("Failed to read EFUSE at line%d! Err code = %X\n", __LINE__, ret);
        return ret;
    }

    first_u64 = *(hi_u64 *)&tmp_data[0];
    second_u8 = *(hi_u8 *)&tmp_data[8]; /* the last u8 bit */
    /* Discard the first u64 multi-read low bit (diff_head_read bit).*/
    first_u64 = first_u64 >> diff_head_read;
    /* Obtain the lower bits of the second char as the upper bits of the first u64 (diff_head_read 
bit). */
    first_u64 = first_u64 | ((hi_u64)second_u8 << (64 - diff_head_read)); /* left shift (64 – 
diff_head_read) bits leftwards */
    *(hi_u64 *)read_data = first_u64;

    printf("usr_data = 0x%08X %08X\n", read_data[0], read_data[1]);

    return HI_ERR_SUCCESS;
}

hi_u32 efuse_usr_write(hi_void)
{
    hi_u32 ret;
    hi_u32 write_data[EFUSE_USR_RW_SAMPLE_BUFF_MAX_LEN] = {
        0x0,
        0x1,
    };
    hi_u16 start_bit = 0x75C;       /* The offset address of customer_rsvd0 is 0x75C. */
    hi_u16 rw_bits = 64;            /* The length of customer_rsvd0 is 64 bits. */

    ret = efuse_usr_read();
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }

    ret = hi_efuse_usr_write(start_bit, rw_bits, (hi_u8 *)write_data);
    if (ret != HI_ERR_SUCCESS) {
        printf("Failed to write EFUSE!\n");
        return ret;
    }

    return HI_ERR_SUCCESS;
}

hi_u32 efuse_usr_lock(hi_void)
{
    hi_u32 ret;
    hi_u8  lock_data = 0x1;
    hi_u16 lock_start_bit = 0x7FD;  /* The lock offset address of customer_rsvd0 is 0x7FD. */
    hi_u16 lock_bits = 1;           /* The lock length of customer_rsvd0 is 1 bit. */

    efuse_get_lock_stat();

    ret = hi_efuse_usr_write(lock_start_bit, lock_bits, &lock_data);
    if (ret != HI_ERR_SUCCESS) {
        printf("Failed to lock EFUSE!\n");
        return ret;
    }

    efuse_get_lock_stat();

    return HI_ERR_SUCCESS;
}

hi_u32 sample_usr_efuse(hi_void)
{
    hi_u32 ret;

#ifdef EFUSE_WRITE_ENABLE
    ret = efuse_usr_write();
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }
#endif

    ret = efuse_usr_read();
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }

#ifdef EFUSE_LOCK_ENABLE
    ret = efuse_usr_lock();
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }
#endif

    return HI_ERR_SUCCESS;
}
```