How to use system calls (SROM APIs) for eFuse handling in TRAVEO™ T2G family

About this document

Scope and purpose

This application note provides useful information for eFuse memory handling using system calls in TRAVEO™ T2G MCUs. The eFuse memory consists of a set of eFuse bits. The eFuse bits are used to store various unchanging device parameters. System calls (SROM APIs) are available to program and read eFuses. This document describes how to use System calls for eFuse handling, and points to note.

Intended audience

This document is intended for anyone using the TRAVEO™ T2G family.

Table of contents

1 Introduction

TRAVEO™ T2G family MCUs include Arm® Cortex®-M CPUs, SRAM, flash memories, hardware-based Cryptography (eSHE/HSM support), communication peripherals, digital peripherals, and analog peripherals in a single chip. In addition, the MCUs contain 1024 bits One-Time-Programmable (OTP) eFuse memory that can be used to store and access a unique and unalterable identifier or serial number for each device.

When an eFuse bit is programmed, or "blown", its value cannot be changed. Some of the eFuse bits are used to store various unchanging device parameters, including critical device factory trim settings, device lifecycle stages, DAP security settings, and encryption keys. Other eFuse bits are available for customer use. For more information, see the "eFuse memory" section in the TRAVEO™ T2G architecture reference manual [2].

Customers may need to program and bloweFuse to control the device lifecycle stage. System calls are used to read and program eFuse.

2 System call

TRAVEO™ T2G device has a supervisory ROM that contains Boot ROM code and SROM APIs. The SROM APIs are executed by Arm® Cortex®-M0+ (CM0+). The system calls are used for specific operations, such as flash programming and lifecycle management. For more information, see the "Nonvolatile Memory Programming" in the TRAVEO™ T2G architecture reference manual.

2.1 System call interface

System calls are run via Inter-processor communication (IPC) structures and it can be triggered from CM core (CMx: CM0+, CM4, CM7_0, CM7_1, CM7_2, and CM7_3) or Debug Access Port (DAP) at any point during code execution. System calls should acquire the IPC_STRUCT reserved for them and provide arguments and notify IPC interrupt '0' to trigger a system call. When the API operation is complete, CM0+ will release the IPC structure that initiated the system call. If an interrupt is required upon release, then the corresponding mask bit should be set in IPC_INTR_STRUCT.INTR_MASK.RELEASE[i]. For a list of lists IPC_STRUCTs reserved for CMx and DAP, see Table 1. For system call interface for CYT4BF series, see Figure 1. For other series, see the "Nonvolatile Memory Programming" section in TRAVEO™ T2G architecture reference manual [2].

Table 1 **IPCSTRUCT for system call interface**

Number of IPC Structure	CYT2 series	CYT3/4 series	CYT6 series
IPC_STRUCT0	CM0+	CM0+	CM0+
IPC_STRUCT1	CM4	CM7_0	CM7_0
IPC_STRUCT2	DAP	CM7_1 (Only CYT4 series), DAP (Only CYT3DL series)	CM7_1
IPC_STRUCT3	Unused	DAP (Excluding CYT3DL series), Unused (Only CYT3DL series)	CM7_2
IPC_STRUCT4	Unused	Unused	CM7_3
IPC_STRUCT5	Unused	Unused	DAP

Note: *When the system call requests are received from multiple masters at the same time, priority is given to the IPC structures in order of their number. Specifically, CM0+ has the highest priority and DAP has the lowest priority.*

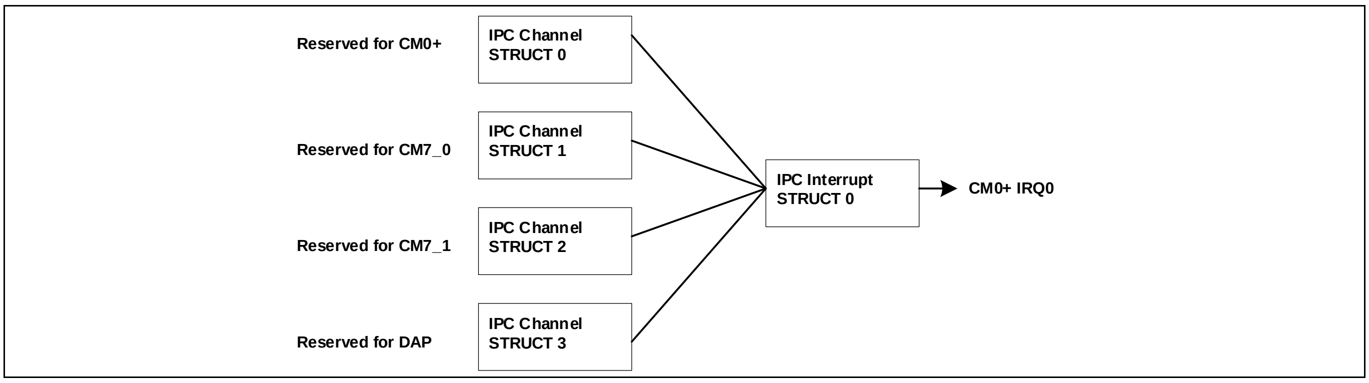

Figure 1 **System call interface for CYT4BF series**

2.2 System call argument

The IPC component implements two 32-bit data registers, but only one (IPC_STRUCTx_DATA0) of these two registers is used to pass parameters to the system calls. The value of "x" depends on the master calling the API. This argument is either a pointer to SRAM or a formatted opcode or argument value that cannot be a valid SRAM address.

Note: *Exceptionally, the ConfigureRegulator system call uses IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1.*

Following is the difference between CMx and DAP encoding:

- DAP: If the sum of the opcode and the argument (opcode + argument) is less than or equal to 31 bits, store them in the data field and set the LSB of the data field as '1'. When system calls are completed, a return value is passed in the IPC data register. For calls that need more argument data, the data field is a pointer to a structure in SRAM that has the opcode and the argument. The pointer needs to align on a word boundary. Therefore, it is a pointer if and only if the LSB is 0. In this case, when system calls are completed, a return value is passed to pointer address in the SRAM.
- CMx: A pointer is always used to a structure in SRAM. Commands that are issued as a single word by DAP can still be issued by CMx but use an SRAM structure instead.

For argument of each system call, see the "Nonvolatile Memory Programming" in TRAVEO™ T2G architecture reference manual [2].

Some system calls cannot be called from either CMx or DAP, depending on the chip protection state. For more information, see the "SROM API Library" section in TRAVEO™ T2G architecture reference manual [2].

2.3 Verification of system call

At the end of every system call, a status code is written over the arguments in the IPC data register, or the SRAM address pointed by the IPC data register (determined by the placement of the arguments).

When a system call is successful, the API writes 0xAxxxxxxx, where "x" indicates a "don't care" value or return value from the system call. If the system call is a failure, then the API writes 0xF00000X, where "X" indicates the failure code.

For more information on the failure code, see the "System Call Status" in TRAVEO™ T2G architecture reference manual [2].

3 System calls for eFuse handling

This section describes system calls related to eFuse programming, reading, and device lifecycle management.

Table 2 lists for system calls for eFuse handling. For more information, see the "Nonvolatile Memory Programming" in TRAVEO™ T2G architecture reference manual [2].

Table 2 List of system calls for eFuse handling

System call	Access Allowed (Protection state)		
	Normal	**Secure**	**Dead**
BlowFuseBit	CMx, DAP	CMx	–
ReadFuseByte	CMx, DAP	CMx, DAP	–
ReadFuseByteMargin	CMx, DAP	CMx, DAP	–
TransitiontoSecure	CMx, DAP	–	–
TransitiontoRMA	–	CMx, DAP	–

Note: *There are other system calls also that could read eFuse, for example, CheckFactoryHash. See the "Nonvolatile Memory Programming" in TRAVEO™ T2G architecture reference manual [2] for details of each system.*

3.1 BlowFuseBit

This system call blows the addressed eFuse bit. The read value of a blown eFuse bit is '1'. The parameters and results of the BlowFuseBit system call are described in the following tables.

For the arguments of the BlowFuseBit system call, see Table 3 and Table 4. The return result after running the system call is shown in Table 5.

Table 3 Argument located in IPC_STRUCTx_DATA0

Address field	Name and value	Description
IPC_STRUCTx_DATA0		
Bits [31:24]	0x01	Opcode
Bits [23:16]	Byte Address	A row number within a column
Bits [15:12]	Macro Address	A column number
Bits [11]	Not used	-
Bits [10:8]	Bit Address	A bit to be blown in the selected byte
Bits [7:1]	Not used	–
Bits [0]	0x1	Indicates that the arguments are passed in the DATA0 register

Table 4 Argument located in SRAM

Address field	Name and value	Description
IPC_STRUCTx_DATA0		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address. Therefore, Bits [1:0] is "0"
SRAM_SCRATCH_ADDR		
Bits [31:24]	0x01	Opcode
Bits [23:16]	Byte Address	A row number within a column
Bits [15:12]	Macro Address	A column number
Bits [11]	Not used	–
Bits [10:8]	Bit Address	A bit to be blown in the selected byte. Value in the range [0,7]
Bits [7:0]	Not used	–

Table 5 Return result

Address field	Name and value	Description
IPC_STRUCTx_DATA0 or SRAM_SCRATCH_ADDR (depends on Argument location)		
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	System call result
Bits [27:24]	Not used	–
Bits [23:0]	Error code	Error code if status is error, otherwise don't care. For error code, see the "System Call Status" in TRAVEO™ T2G architecture reference manual [2]

3.2 ReadFuseByte

This system call returns the value of an eFuse. The read value of a blown eFuse bit is '1'and that of a not blown eFuse bit is '0'. This API inherits the client protection context.

For the arguments of the ReadFuseByte system call, see Table 6 and Table 7. The return result after running the system call is shown in Table 8.

Table 6 Argument located in IPC_STRUCTx_DATA0

Address field	Name and value	Description
IPC_STRUCTx_DATA0		
Bits [31:24]	0x03	Opcode
Bits [23:8]	eFuse address	eFuse byte address to read from
Bits [7:0]	0x01	Indicates that the arguments are passed in the DATA0 register

Table 7 Argument located in SRAM

Address field	Name and value	Description
IPC_STRUCTx_DATA0		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address. Therefore, Bits [1:0] is 0
SRAM_SCRATCH_ADDR		
Bits [31:24]	0x03	Opcode
Bits [23:8]	eFuse address	eFuse byte address to read from
Bits [7:0]	Not used	–

Table 8 Return result

Address field	Name and value	Description
IPC_STRUCTx_DATA0 or SRAM_SCRATCH_ADDR (depends on Argument location)		
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Result of System call
Bits [27:24]	Not used	–
Bits [23:0]	eFuse byte read value or error code	Read value when status is success, otherwise error code. For error code, see the “System Call Status” in TRAVEO™ T2G architecture reference manual [2]

3.3 ReadFuseByteMargin

This system call returns the eFuse contents of the addressed byte read marginally. The read value of a blown eFuse bit is '1' and that of not blown is '0'. Difference between ReadFuseByte and ReadFuseByteMargin is that ReadFuseByte always uses default read condition (Margin control = “1”). This API inherits client's protection context.

For the arguments of the ReadFuseByteMargin system call, see Table 9 and Table 10. The return result after running the system call is shown in Table 11.

Table 9 Argument located in IPC_STRUCTx_DATA0

Address field	Name and value	Description
IPC_STRUCTx_DATA0		
Bits [31:24]	0x2B	Opcode
Bits [23:20]	0: Low resistance (-50% from nominal) 1: Nominal resistance (default read condition) 2: High resistance (+50% from nominal)	Margin control for read-back check: 0: eFuse is read marginally with very low resistance after the eFuse is blown. A read pass is not an indication that the eFuse is blown

Address field	Name and value	Description
	Other: Higher resistance (+100% from nominal)	strongly and will be stable over lifetime 1: eFuse is read marginally with nominal resistance after the eFuse is blown. This is the default read, also used by ReadFuseByte API. A read pass is not an indication that the eFuse is blown strongly and will be stable over lifetime 2: eFuse is read marginally with high resistance after the eFuse is blown. A read pass is an indication that the eFuse is blown strongly and will be stable over lifetime Other: eFuse is read marginally with the highest resistance after the eFuse is blown. A read pass is an indication that the eFuse is blown strongly and will be stable over lifetime Please refer to Software Flow (Figure 4)
Bits [19:8]	eFuse address	eFuse byte address to read from
Bits [7:0]	0x01	Indicates that the arguments are passed in the DATA0 register

Table 10 Argument located in SRAM

Address field	Name and value	Description
IPC_STRUCTx_DATA0		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address. Therefore, Bits [1:0] is 0
SRAM_SCRATCH_ADDR		
Bits [31:24]	0x2B	Opcode
Bits [23:20]	0: Low resistance (-50% from nominal) 1: Nominal resistance (default read condition) 2: High resistance (+50% from nominal) Other: Higher resistance (+100% from nominal)	Margin control for read-back check: See Margin control for read-back check in Table 9 for details

Address field	Name and value	Description
Bits [19:8]	eFuse address	eFuse byte address to read from
Bits [7:0]	Not used	–

Table 11 Return result

Address field	Name and value	Description
IPC_STRUCTx_DATA0 or SRAM_SCRATCH_ADDR (depends on Argument location)		
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Result of System call
Bits [27:24]	Not used	–
Bits [23:0]	eFuse byte read value or error code	Read value when status is success, otherwise error code.

For possible return error codes, see Table 12. Note that this list is not exhaustive. For a comprehensive list of all error codes, see the "System Call Status" in TRAVEO™ T2G architecture reference manual [2].

Table 12 Error code for BlowFuseBit, ReadFuseByte, and ReadFuseByteMargin system call

Return Value	Description
0xAxxxxxXY	Success, where XY = 0x0, 0x1, 0x2,…, 0xFF depending on blown fuse bits in addressed eFuse byte. Where "x" indicates don't care value.
0xF0000001	Invalid protection state – This system call is not available in current protection state
0xF0000002	Invalid eFuse address
0xF0000005	eFuse bytes are read/write protected via protection unit
0xF0000006	Client did not use its reserved IPC structure for invoking system call
0xF0000008	Returned by all system calls when client does not have access to the region it is using to pass arguments
0xF000000B	The opcode is not a valid system call opcode
0xF000000F	Invalid arguments passed to the system call
0xF0000013	Invalid argument location

3.4 TransitiontoSecure

This system call validates the FACTORY_HASH and programs SECURE_HASH, secure access restrictions and dead access restrictions into eFuse. Also, it programs eFuse bit of secure or secure with debug to transition to SECURE or SECURE WITH DEBUG life-cycle stage. Therefore, when this system call is executed successfully, the device will transition to Secure protection state after the next device reset. This system call is only allowed in

Normal protection state, that is, NORMAL_PROVISIONED lifecycle stage. The arguments in this system call can only be in SRAM, and it requires three words for parameters in SRAM.

Table 13 shows the arguments for the TransitiontoSecure system call and Table 14 shows the return result after running system call.

Table 13 Argument located in SRAM

Address field	Name and value	Description
IPC_STRUCTx_DATA0		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address. Therefore, Bits [1:0] is 0
SRAM_SCRATCH_ADDR		
Bits [31:24]	0x2F	Opcode
Bits [15:8]	1: blow D fuse Other: blow S fuse	Decide whether to transition to SECURE or SECURE WITH DEBUG lifecycle stage
Bits [7:0]	Not used	–
SRAM_SCRATCH_ADDR +0x4		
Bits [31:20]	Not used	–
Bits [19:18]	MMIO_ALLOWED	Configure Secure Access Restriction for encoding, see the “eFuse Bits” in TRAVEO™ T2G architecture reference manual [2]
Bits [17:16]	SFLASH_ALLOWED	
Bits [15:14]	WORK_FLASH_ALLOWED	
Bits [13:11]	SRAM_ALLOWED	
Bits [10:8]	FLASH_ALLOWED	
Bits [7]	DIRECT_EXECUTE_DISABLE	
Bits [6]	SYS_AP_MPU_ENABLE	
Bits [5:4]	AP_CTL_SYS_DISABLE	
Bits [3:2]	AP_CTL_CM4_DISABLE	
Bits [1:0]	AP_CTL_CM0_DISABLE	
SRAM_SCRATCH_ADDR +0x8		
Bits [31:0]	See above	Configure Secure Dead Access Restriction. For encoding, see the “eFuse Bits” in TRAVEO™ T2G architecture reference manual [2]

Table 14 Return result

Address field	Name and value	Description
SRAM_SCRATCH_ADDR		
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	System call result

Address field	Name and value	Description
Bits [27:24]	Not used	–
Bits [23:0]	Error code	Error code if status is error, otherwise don't care

Table 15 shows possible return error code. Note that it is not exhaustive. For all error codes, see the "System Call Status" in TRAVEO™ T2G architecture reference manual [2].

Table 15 Error code for TransitiontoSecure system call

Return value	Description
0xAxxxxxxx	Success, where "x" indicates don't care value
0xF0000001	Invalid protection state – This system call is not available in current protection state. For example, TransitiontoSecure triggered when the device is already in Secure
0xF00000B5	Invalid Factory Hash
0xF0000018	TOC1 is invalid
0xF00000B8	TOC1 has incorrect hash objects
0xF00000B9	TOC2 has incorrect hash objects
0xF0000020	TOC2 is invalid
0xF00000CB	eFuse programming failed (only for lifecycle stage eFuse bits).
0xF50000xx	Crypto resources are protected against PC1 access, xx corresponding to protected PPU ID

3.5 TransitiontoRMA

This system call blows the eFuse bit and converts the parts from SECURE or SECURE WITH DEBUG to the RMA lifecycle stage. Therefore, this system call is only allowed in Secure protection state, that is, SECURE or SECURE WITH DEBUG lifecycle stage. This system call requires parameters such as certificate and digital signature to move a device to the RMA lifecycle stage. The arguments in this system call can only be in SRAM, and it requires seven words for the parameters and signature area in SRAM. The signature size depends on the key size used.

Table 16 shows the arguments for the TransitiontoSecure system call.

Table 17 shows the return result after running system call.

Table 16 Argument located in SRAM

Address field	Name and value	Description
IPC_STRUCTx_DATA0		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address. Therefore, Bits [1:0] is 0
SRAM_SCRATCH_ADDR		

Address field	Name and value	Description
Bits [31:24]	0x28	Opcode
Bits [23:0]	Not used	–
SRAM_SCRATCH_ADDR +0x4		
Bits [31:0]	0x14	Object size in bytes (including itself). It should always be 20 bytes.
SRAM_SCRATCH_ADDR +0x8		
Bits [31:0]	0x120028F0	Command ID
SRAM_SCRATCH_ADDR +0xC		
Bits [31:0]	Device unique ID 0	Unique ID word 0
SRAM_SCRATCH_ADDR +0x10		
Bits [31:0]	Device unique ID 1	Unique ID word 1
SRAM_SCRATCH_ADDR +0x14		
Bits [31:0]	Device unique ID 2	Unique ID word 2 (3 bytes + zero padding)
SRAM_SCRATCH_ADDR +0x18		
Bits [31:0]	Signature address	SRAM address where signature is stored
SRAM_SCRATCH_ADDR		
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	System call result
Bits [27:24]	Not used	–
Bits [23:0]	Error code	Error code if status is error, otherwise don't care. For error code, see the "System Call Status" in TRAVEO™ T2G architecture reference manual [2]

Table 17 Return result

Address field	Name and value	Description
SRAM_SCRATCH_ADDR		
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	System call result
Bits [27:24]	Not used	–
Bits [23:0]	Error code	Error code if status is error, otherwise don't care For error code ,see the "System Call Status" in TRAVEO™ T2G architecture reference manual [2]

4 eFuse structure

eFuse consists of multiple eFuse macros and some of them are available for customer use.

Figure 2 shows eFuse structure in TRAVEO™ T2G series. It has four eFUSE macros and a 192-bit (24-byte) customer data area (Offset= 0x68 to 0x7F). Customers can read and program this area in the eFuse using the BlowFuseBit, ReadFuseByte, and ReadFuseBytesMargin system calls. For more information on the number of eFuse macros and customer data, see the “Peripheral I/O map” in the device datasheet [1].

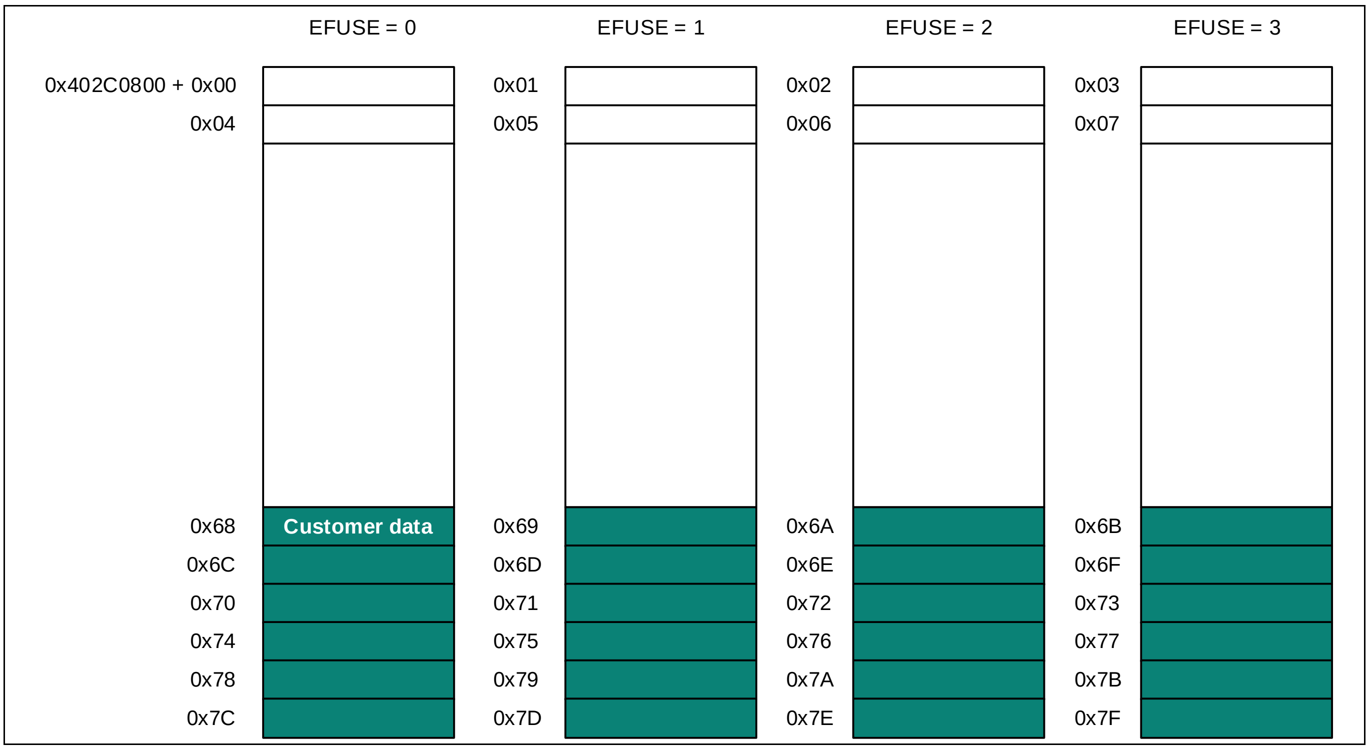

Figure 2 eFuse structure in CYT4BF series

4.1 How to specify the parameters for BlowFuseBit system call

A BlowFuseBit system call is used to program specified eFuse bits.

As parameters, it has:

- Macro address: Specifies the eFuse macro number to program
- Byte address: Specifies the eFuse byte number to program in the eFuse macro
- Bit address: Specifies the bit position to program in the eFuse byte

Figure 3 shows each parameter description in the BlowFuseBit system call.

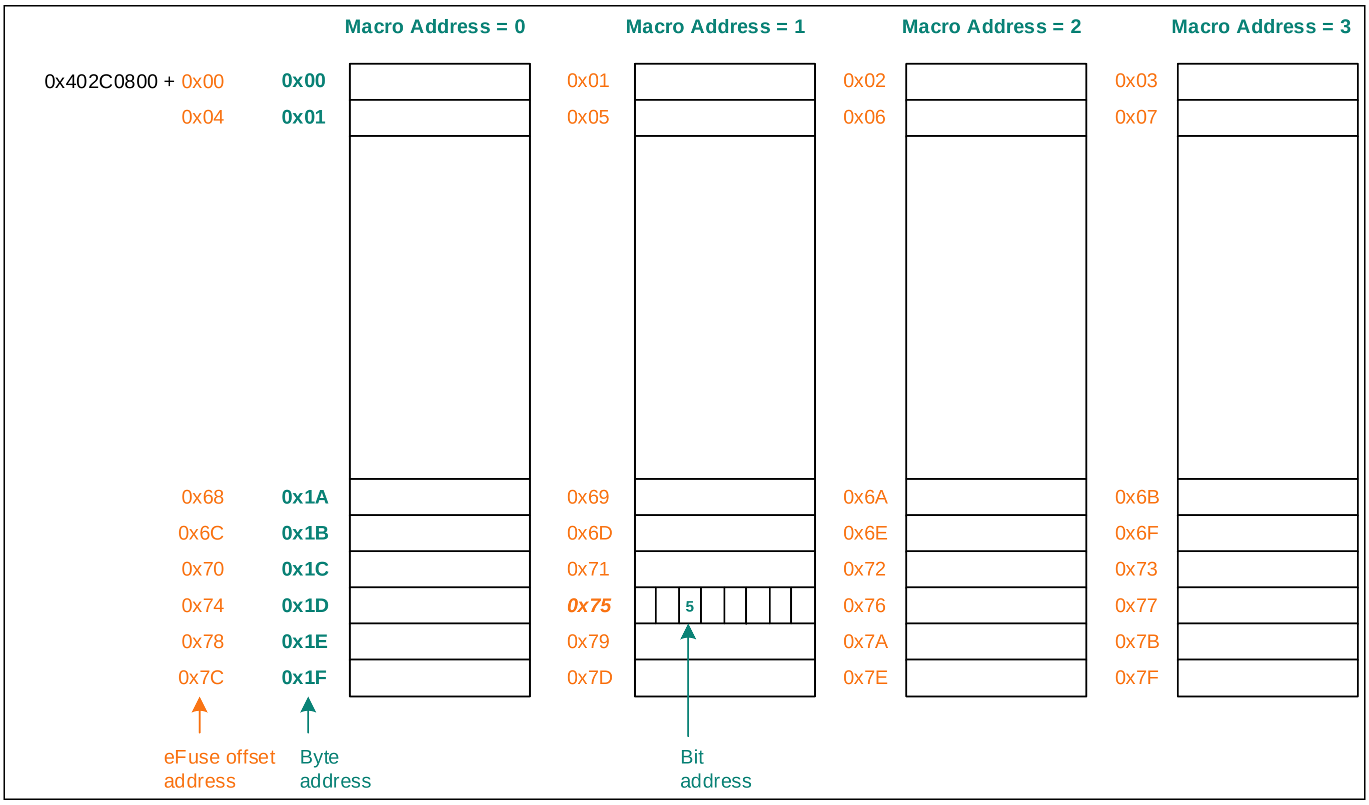

Figure 3 **Parameters in BlowFuseBit**

These parameters have the following relationships:

```
Macro Address = AddressOffset % EFUSE_NR
Byte Address = AddressOffset / EFUSE_NR
```

The variable EFUSE_NR represents the number of macros, which is four for all the T2G devices, while the AddressOffset is the eFuse offset address.

When blowing bit 5 of offset address 0x75 using the BlowFuseBit system call, specify the following parameters:

- Macro address = 0x75% 4 = “1”
- Byte address = 0x75/4 = “29” (0x1D)
- Bit address: “5”

4.2 How to specify the parameters for ReadFuseByte and ReadFuseByteMargin system calls

ReadFuseByte and ReadFuseByteMargin system calls are used to read the eFuse byte.

As parameters, these have:

- eFuse address: Specifies the eFuse byte to read

This parameter specifies the eFuse offset address. Therefore, it is specified one from 104 (0x68) to 127 (0x7F) to read customer data.

5 System requirements to use system calls for eFuse handling

Following are the requirements to run the system calls correctly:

5.1 Power supply

To program an eFuse, ensure the device is in a 'quiet' state with minimal activity and provide a specific power supply. For more information, see the SID40A section in the device datasheet [1].

5.2 Clock configuration

System calls that target blowing of eFuses, such as BlowFuseBit and TransitionToSecure, have some requirements for clk_hf0. To avoid complications, these APIs can be triggered with any of the clock settings used by internal boot (specified by TOC2_FLAGS.CLOCK_CONFIG) before the application changes the clk_hf0 settings.

If the application changes the configurations used by boot for clk_hf0, then ensure that the source clock for clk_hf0 is FLL and the frequency of clk_hf0 is restricted to 100 MHz maximum. If clk_hf0 is not sourced from FLL and FLL is disabled, then the maximum frequency of clk_hf0 should only be 8 MHz.

5.3 Protection Unit

eFuse can be protected by ERPU and EWPU in SWPU. Read or write access using system calls to protected areas returns a failure status such as 0xF0000005. For more information, see AN228680 - Secure system configuration in TRAVEO™ T2G family and KBA234634 - Configure Software Protection Unit (SWPU) objects in TRAVEO™ T2G devices.

5.4 Prerequisites of triggering TransitiontoRMA system call

Requirements to run TransitiontoRMA:

- A device can only run the TransitiontoRMA system call in SECURE or SECURE_WITH_DEBUG lifecycle stage. Therefore, only devices in the secure protection state can trigger this system call, while devices in normal or dead protection states cannot.
- When a device is in SECURE lifecycle stage, a public key is required to be stored on a device to run the TransitiontoRMA system call. After reset, the device waits for OpenRMA system call. When a device transitions to the RMA lifecycle stage from the SECURE_WITH_DEBUG lifecycle stage, OpenRMA is skipped, and the device does not wait for OpenRMA system call.

ECC errors might occur during the TransitiontoRMA system call execution. Therefore, user software should not configure the fault structure for Crypto and SRAM0 ECC errors before triggering the TransitiontoRMA system call. For the fault numbers, see the device and for TransitiontoRMA system call. In addition, see the “System Calls” in TRAVEO™ T2G architecture reference manual [2].

6 How to use system calls

This section describes how to trigger system calls.

In addition, this section describes sample software with the sample driver library (SDL). The code snippets in this application note are part of SDL. For more information on the SDL, see the References Section.

The SDL has two parts: configuration and driver. The configuration part configures the parameter values for the desired operation, whereas the driver part configures each register based on the parameter values in the configuration part. You can configure the configuration part according to your system. The sample software shows the CYT4BF series.

6.1 Blowing eFuse

As mentioned earlier, eFuse has a customer area that customers can use to store the irreversible data. This section describes how to blow eFuse bit for customer area. BlowFuseBit system call is used to blow eFuse bit, and ReadFuseByte and ReadFuseByteMargin system calls are used to confirm the data is written correctly.

6.1.1 Use case

The following shows use case of this code example:

- Clock configuration: CLK_HF0 is configured to be 100 MHz coming from FLL.
- Write address: offset address = 0x6D
- Write data: 0x5A

Where, EFUSE_NR is four for CYT4BF. Therefore, system call parameters used in BlowFuseBit can be calculated as below:

- Macro address: 1 = 109 (0x6D)% 4
- Byte address: 27 = 109 (0x6D)/4
- Bit address: 6, 4, 3, 1 (0x5A)

In this use case, BlowFuseBit system call requires 4 times to blow 4 bits.

6.1.2 Software flow

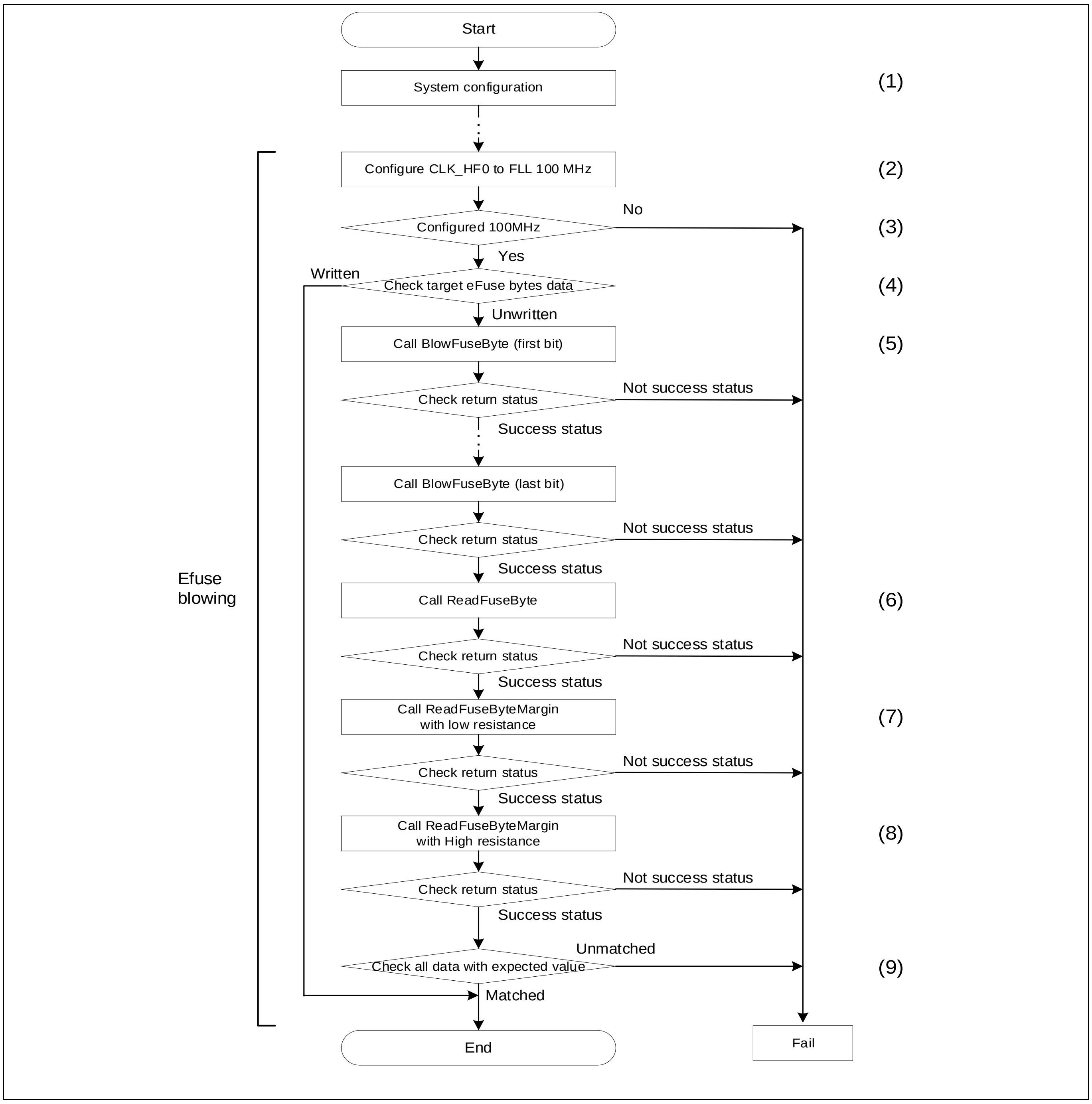

Figure 4 **Blow eFuse flow**

Note: *If a failure occurs, you should handle the error accordingly.*

Note: *In case of failure, it is possible to repeat the eFuse blowing.*

6.1.3 Configuration

For a list of parameters and functions for sample software of blowing eFuse, see Table 18 and Table 19.

Table 18 List of parameters

Parameters	Description	Value
CLK_FREQ_IMO	IMO Frequency	8000000 ul
FLL_TARGET_FREQ	FLL Target Frequency	100000000 ul
WAIT_FOR_STABILIZATION	Wait definition for stabilization time of FLL bypass max switching	10000 ul 333
MacroAddr	Macro Address to blow on BlowFusebit system call	1
ByteAddr	Byte Address to blow on BlowFusebit system call	27
BitAddr	Bit address to blow on BlowFusebit system call	0 (initial value)
MarginCTL	Margin control setting on ReadFuseByteMargin system call	1 (initial value)
EfuseAddr	Offset address of eFuse to read on ReadFuseByte and ReadFuseByteMargin system calls	109
g_RdEfuseValue	Read data by ReadFuseByte system call	0
g_RdEfuseMarginValue50	Read data by ReadFuseByteMargin (-50%) system call	0
g_RdEfuseMarginValue100	Read data by ReadFuseByteMargin (100%) system call	0
g_ExpectedValue	Data to write by BlowFuseBit system call	0x5A

Table 19 List of functions

Functions	Description	Value
ClockSetTo100MHz()	Configure clock to FLL 100 MHz	–
Is_CLK100MHz()	Check clock frequency	–
Cy_BlowFuseBit()	Run the BlowFuseBit system call	–
Cy_ReadFuseByte()	Run the ReadFuseByte system call	–
Cy_ReadFuseByteMargin()	Run the ReadFuseByteMargin system call	–

The following code show an example of running the BlowFuseBit system call with a specific use case:

Code listing 1 Sample software of blowing eFuse

```
#include "cy_project.h"
#include "cy_device_headers.h"
#include <stdio.h>

/** IMO Frequency **/
#define CLK_FREQ_IMO                    (8000000ul)
/** FLL Target Frequency **/
#define FLL_TARGET_FREQ                 (100000000ul)
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION          (10000ul)

/* API Parameters setting*/
uint32_t MacroAddr      = 1;    /* Macro Address = AddressOffset % EFUSE_NR : 0x6D % 4 = '1' */
uint32_t ByteAddr       = 27;   /* Offset = 0x6D (Byte Address = AddressOffset / EFUSE_NR : 0x6D / 4 = '27')*/
uint32_t BitAddr        = 0;    /* It will be update in main code */
uint32_t MarginCTL      = 1;    /* Default read condition: Nominal resistance */
uint32_t EfuseAddr      = 109;  /* 0x0000006D */

/* Global Variable */
uint32_t g_RdEfuseValue = 0;
uint32_t g_RdEfuseMarginValue50 = 0;
uint32_t g_RdEfuseMarginValue100 = 0;
uint32_t g_ExpectedValue = 0x5A;

void ClockSetTo100MHz(void);
bool Is_CLK100MHz(void);
cy_en_srom_driver_status_t Cy_BlowFuseBit(uint32_t blowEfuse_bitaddr, uint32_t blowEfuse_macroaddr, uint32_t
blowEfuse_byteaddr);
cy_en_srom_driver_status_t Cy_ReadFuseByte(uint32_t ReadEfuse_Efuseaddr);
cy_en_srom_driver_status_t Cy_ReadFuseByteMargin(uint32_t Margin_efuseAddr, uint32_t Margin_Ctl);


int main(void)
{
    __enable_irq();

    /*** Initial clock setting for application ***/
    SystemInit();                                                                     (1)

    /*** Disable D-cache if need ***/

    /*** Set the CLK as FLL = 100MHz ***/
    ClockSetTo100MHz();                                                               (2) See Code listing 5

    /*** Check CLK as expected ***/
    while(Is_CLK100MHz() != true);                                                    (3) See Code listing 6

    /*** Read the value of Efuse byte befor write ***/
    while(Cy_ReadFuseByte(EfuseAddr) != CY_SROM_DR_SUCCEEDED);
    if(g_RdEfuseValue == 0)
    {                                                                                 (4) See Code listing 3
        cy_en_srom_driver_status_t Result;

        /***  Call API BlowFuseBit     ***/
        /* Write Efuse bit value as 0x5A (0101 1010) */
        BitAddr = 1;
        Result = Cy_BlowFuseBit(BitAddr, MacroAddr, ByteAddr);                        (5) See Code listing 3
        while(Result != CY_SROM_DR_SUCCEEDED);

        BitAddr = 3;
        Result = Cy_BlowFuseBit(BitAddr, MacroAddr, ByteAddr);
        while(Result != CY_SROM_DR_SUCCEEDED);

        BitAddr = 4;
        Result = Cy_BlowFuseBit(BitAddr, MacroAddr, ByteAddr);
        while(Result != CY_SROM_DR_SUCCEEDED);

        BitAddr = 6;
        Result = Cy_BlowFuseBit(BitAddr, MacroAddr, ByteAddr);
        while(Result != CY_SROM_DR_SUCCEEDED);

        /***  Call API ReadFuseByte     ***/
        Result = Cy_ReadFuseByte(EfuseAddr);                                          (6) See Code listing 3
        while(Result != CY_SROM_DR_SUCCEEDED);

        /***  Call API ReadFuseByteMargin MarginCTL = -50%     ***/                    (7) See Code listing 4
        MarginCTL = 0; /* 0: Low resistance -50% */
        Result = Cy_ReadFuseByteMargin(EfuseAddr, MarginCTL);
        while(Result != CY_SROM_DR_SUCCEEDED);
                                                                                      (8) See Code listing 4
        /***  Call API ReadFuseByteMargin MarginCTL = +100%     ***/
        MarginCTL = 3; /* High resistance +100% */
        Result = Cy_ReadFuseByteMargin(EfuseAddr, MarginCTL);
        while(Result != CY_SROM_DR_SUCCEEDED);

        if((g_ExpectedValue != g_RdEfuseValue)||(g_ExpectedValue !=
            g_RdEfuseMarginValue50)||(g_ExpectedValue != g_RdEfuseMarginValue100))    (9)
        {
            /* mismattch */
            while(1);
        }
    }
    else
    {
        while(1);
    }

    /*** Switch to initial clock setting for application ***/

    for(;;);

}
```

Code listing 2 Cy_BlowFuseBit function

```
cy_en_srom_driver_status_t Cy_BlowFuseBit(uint32_t blowEfuse_bitaddr, uint32_t blowEfuse_macroaddr, uint32_t
blowEfuse_byteaddr)
{
    /* Prepares arguments to be passed to SROM API */
    un_srom_api_args_t apiArgs  = { 0ul };
    apiArgs.BlowFuseBit.arg0.bitAddr    = blowEfuse_bitaddr;
    apiArgs.BlowFuseBit.arg0.macroAddr  = blowEfuse_macroaddr;
    apiArgs.BlowFuseBit.arg0.byteAddr   = blowEfuse_byteaddr;
    apiArgs.BlowFuseBit.arg0.Opcode     = CY_SROM_OP_BLOW_FUSE_BIT;

    /* Call SROM API driver and process response */
    while(Cy_Srom_CallApi(&apiArgs, NULL) != CY_SROM_DR_SUCCEEDED);

    return CY_SROM_DR_SUCCEEDED;
}
```

Code listing 3 Cy_ReadFuseByte function

```
cy_en_srom_driver_status_t Cy_ReadFuseByte(uint32_t ReadEfuse_Efuseaddr)
{
    /* Prepares arguments to be passed to SROM API */
    un_srom_api_args_t apiArgs  = { 0ul };
    un_srom_api_resps_t apiResp = { 0ul };
    apiArgs.RdFuse.arg0.eFuseAddr   = ReadEfuse_Efuseaddr;
    apiArgs.RdFuse.arg0.Opcode      = CY_SROM_OP_READ_FUSE_BYTE;

    /* Call SROM API driver and process response */
    while(Cy_Srom_CallApi(&apiArgs, &apiResp) != CY_SROM_DR_SUCCEEDED);

    g_RdEfuseValue = apiResp.RdFuse.resp0.ReadByte;

    return CY_SROM_DR_SUCCEEDED;
}
```

Code listing 4 Cy_ReadFuseByteMargin function

```
cy_en_srom_driver_status_t Cy_ReadFuseByteMargin(uint32_t Margin_efuseAddr, uint32_t Margin_Ctl)
{
    /* Prepares arguments to be passed to SROM API */
    un_srom_api_args_t apiArgs = { 0ul };
    un_srom_api_resps_t apiResp = { 0ul };
    apiArgs.RdFuseMargin.arg0.eFuseAddr     = Margin_efuseAddr;
    apiArgs.RdFuseMargin.arg0.marginCtl     = Margin_Ctl;
    apiArgs.RdFuseMargin.arg0.Opcode        = CY_SROM_OP_READ_FUSE_BYTE_MARGIN;

    /* Call SROM API driver and process response */
    while(Cy_Srom_CallApi(&apiArgs,&apiResp) != CY_SROM_DR_SUCCEEDED);

    if(MarginCTL == 0 ) /* 0: Low resistance -50% */
    {
        g_RdEfuseMarginValue50 = apiResp.RdFuseMargin.resp0.ReadByte;
    }
    else if(MarginCTL == 3)/* Higher resistance(+100% from nominal) */
    {
        g_RdEfuseMarginValue100 = apiResp.RdFuseMargin.resp0.ReadByte;
    }
    else
    {
       /* MarginCTL parameter error */
       while(1);

    }
    return CY_SROM_DR_SUCCEEDED;
}
```

Code listing 5 ClockSetTo100MHz function

```
void ClockSetTo100MHz(void)
{
    /* Disable Fll */
    CY_ASSERT(Cy_SysClk_FllDisableSequence(WAIT_FOR_STABILIZATION) == CY_SYSCLK_SUCCESS);
    /* Wait */
    for(volatile uint32_t I = 0ul; i < 3000; i++);

    /* CLK_MEM */
    CPUSS->unMEM_CLOCK_CTL.stcField.u8INT_DIV     = 0u; /* no division */

    /* CLK_SLOW */
    CPUSS->unSLOW_CLOCK_CTL.stcField.u8INT_DIV    = 0u; /* no division */

    /***** FLL(PATH0) source setting ******/
    Cy_SysClk_ClkPathSetSource(0, CY_SYSCLK_CLKPATH_IN_IMO);
    CY_ASSERT(Cy_SysClk_HfClockSetDivider(CY_SYSCLK_HFCLK_0, CY_SYSCLK_HFCLK_NO_DIVIDE) == CY_SYSCLK_SUCCESS);
    CY_ASSERT(Cy_SysClk_HfClockSetSource(CY_SYSCLK_HFCLK_0,
(cy_en_hf_clk_sources_t)CY_SYSCLK_HFCLK_IN_CLKPATH0) == CY_SYSCLK_SUCCESS);
    CY_ASSERT(Cy_SysClk_HfClkEnable(CY_SYSCLK_HFCLK_0) == CY_SYSCLK_SUCCESS);

    /* Enable Fll */
    CY_ASSERT(Cy_SysClk_FllEnable(WAIT_FOR_STABILIZATION) == CY_SYSCLK_SUCCESS);
}
```

Code listing 6 Is_CLK100MHz function

```
{
     cy_stc_base_clk_freq_t freqInfo =
    {
        .clk_imo_freq  = CY_CLK_IMO_FREQ_HZ,
        .clk_ext_freq  = 0,
        .clk_eco_freq  = CY_CLK_ECO_FREQ_HZ,
        .clk_ilo0_freq = CY_CLK_HVILO0_FREQ_HZ,
        .clk_ilo1_freq = CY_CLK_HVILO1_FREQ_HZ,
        .clk_wco_freq  = CY_CLK_WCO_FREQ_HZ,
    };

    uint32_t coreClockFreq = 0;
    uint32_t Hf0ClkFrequency = 0;

    Cy_SysClk_InitGetFreqParams(&freqInfo);
    Cy_SysClk_GetHfClkFrequency(CY_SYSCLK_HFCLK_0,&Hf0ClkFrequency);
    Cy_SysClk_GetCoreFrequency(&coreClockFreq);

    if((coreClockFreq == FLL_TARGET_FREQ) && (Hf0ClkFrequency == FLL_TARGET_FREQ))
    {
        return true;
    }

    return false;
}
```

6.2 Transition to Secure protection state

This section describes how to transition a device to the SECURE lifecycle stage. The TransitiontoSecure system call transitions the device's lifecycle stage to SECURE or SECURE_WITH_DEBUG. This system call validates the FACTORY_HASH and programs the SECURE HASH to eFuse. In addition, when the device transitions to the SECURE lifecycle stage, this system call programs Secure and Dead access restrictions into eFuse. For Secure and Dead access restrictions encoding, see the "BootROM" section in TRAVEO™ T2G architecture reference manual [2].

6.2.1 Use case

The following shows use case of this code example:

- Transition to SECURE lifecycle stage
- Secure access restrictions: 0x00000000
- Dead access restrictions: 0x00000000

Secure and Dead access restriction will change according to security requirements.

6.2.2 Software flow

Figure 5 to Figure 7 shows the software flow of blowing eFuse.

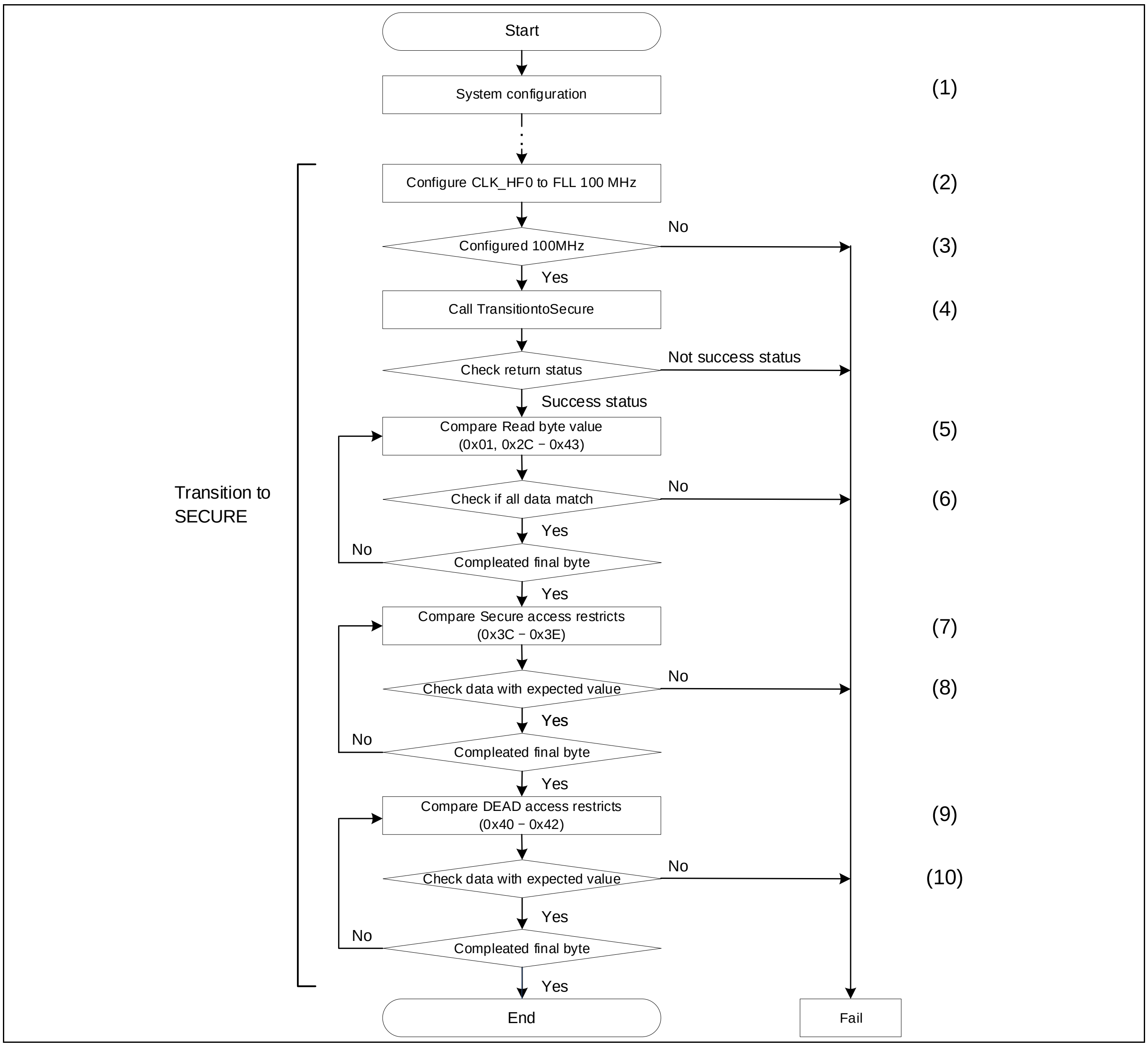

Figure 5 Transition to SECURE lifecycle stage flow

Note: If a failure occurs, you should handle the error accordingly.

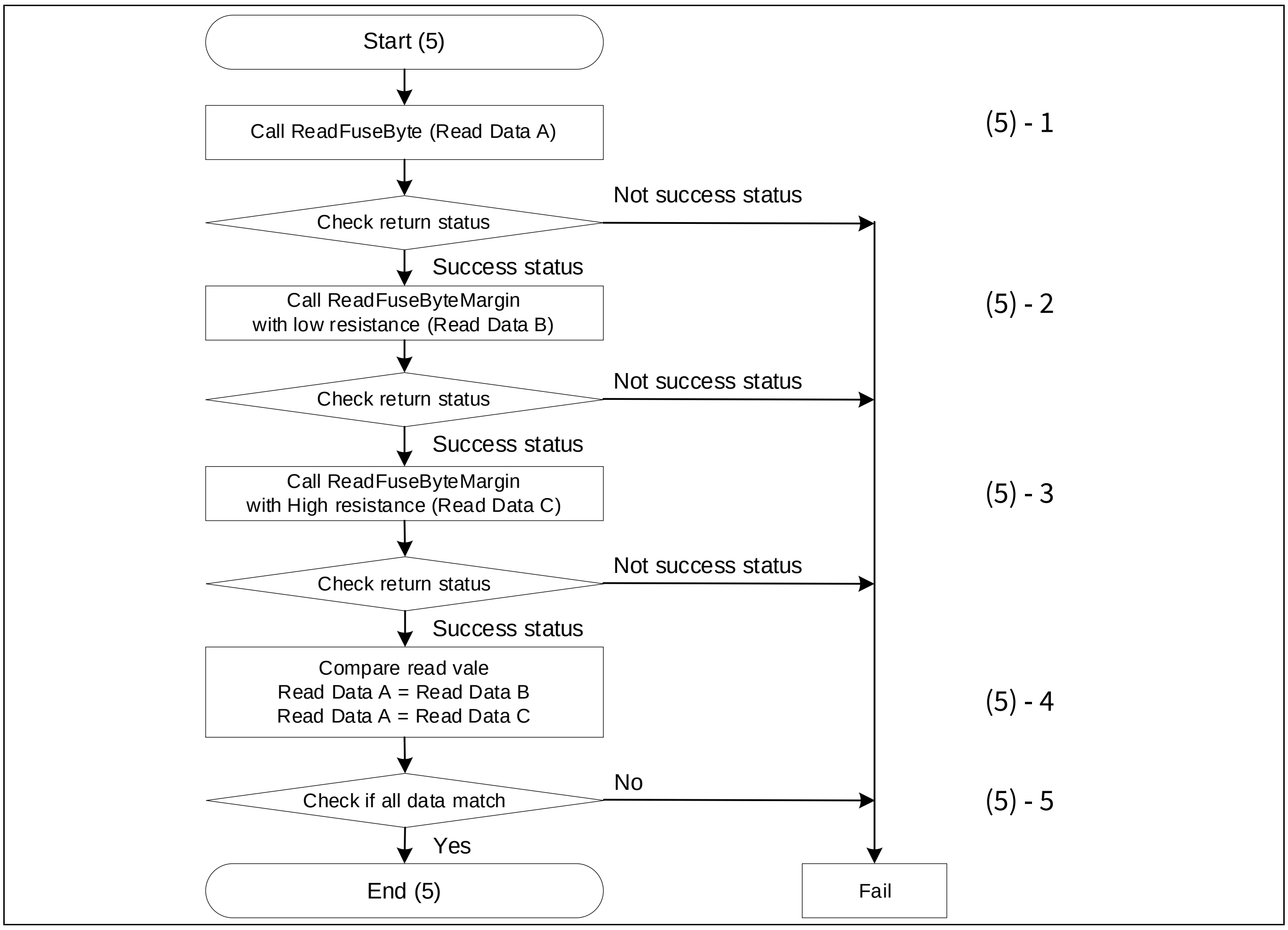

Figure 6 **Compare Read byte value process (5) flow**

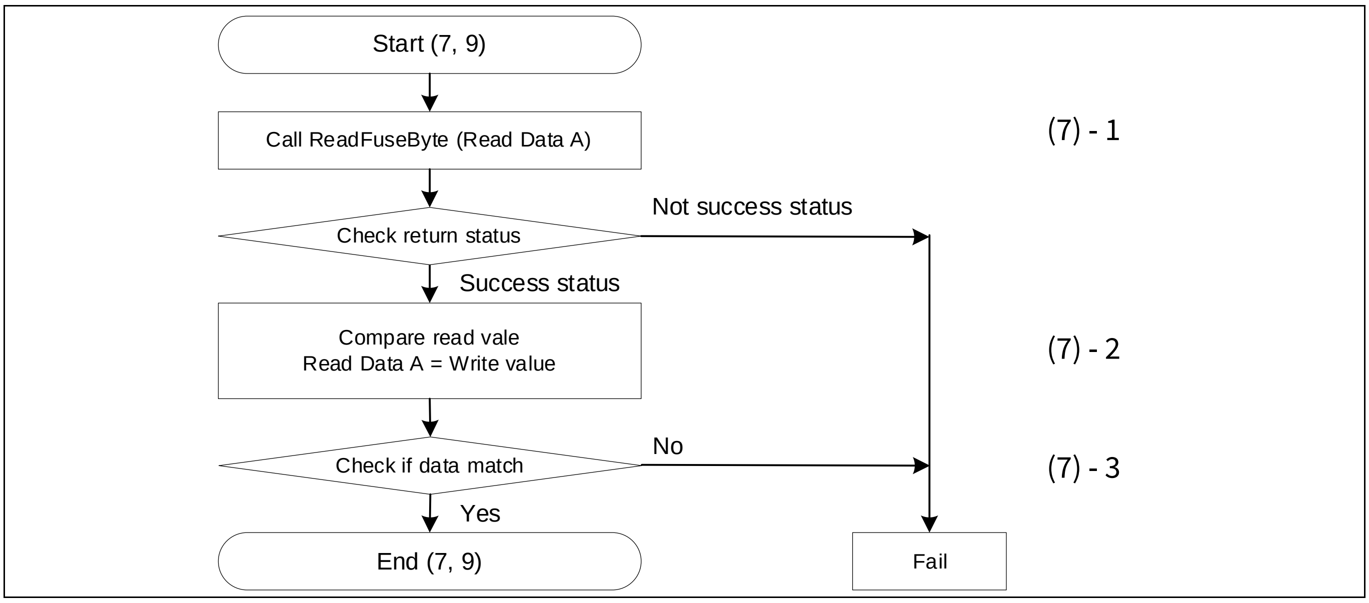

Figure 7 **Compare Secure and DEAD access restricts process (8), (11) flow**

Note: *Some requirements should be fulfilled before transitioning the device to SECURE or SECURE_WITH_DEBUG lifecycle stage such as programming a valid public key, using Cypress* *secure application format (CySAF) etc. For more details, see AN228680 - Secure system configuration in TRAVEO™ T2G family.*

Note: *If all the steps described in the flow chart is successful, then it is recommended to do a device reset and make sure that the device is in the intended lifecycle stage (For example SECURE). If ROM boot reaches SECURE lifecycle stage, then the lifecycle eFuse and the secure hash must be correct.*

6.2.3 Configuration

For a list of parameters and functions for sample software of blowing eFuse, see Table 18 and Table 19.

Table 20 List of parameters

Parameters	Description	Value
CLK_FREQ_IMO	IMO Frequency	8000000 ul
FLL_TARGET_FREQ	FLL Target Frequency	100000000 ul
WAIT_FOR_STABILIZATION	Wait definition for stabilization time of FLL bypass max switching	10000ul
CY_SROM_OP_TRANSITION_TO_SECURE	TransitiontoSecure system call OP code	0x2F
SECURE	TransitiontoSecure system call parameter: Transition to SECURE lifecycle stage:	0x0
SECURE_ACC_RESTRICT	Secure access restricts data. See Table 13 for configuration. This parameter can change according to system security requirements.	0x00000000
DEAD_ACC_RESTRICT	DEAD access restricts data. See Table 13 for configuration. This parameter can change according to system security requirements.	0x00000000
MarginCTL	ReadFuseByteMargin system call parameter.	1
EfuseAddr[]	eFuse Address	1, 44-67
SecureAccessRestrictions_EfuseAddr[]	eFuse Address of Secure access restriction	60-62
DeadAccessRestrictions_EfuseAddr[]	eFuse Address of DEAD access restriction	64-66
g_RdEfuseValue	Read data by ReadFuseByte system call	–
g_RdEfuseMarginValue50	Read data by ReadFuseByteMargin (-50%) system call	–

Parameters	Description	Value
g_RdEfuseMarginValue100	Read data by ReadFuseByteMargin (100%) system call	–

Table 21 List of functions

Functions	Description	Value
ClockSetTo100MHz()	Configure clock to FLL 100 MHz	–
Is_CLK100MHz()	Check clock frequency	–
CompareEfuseValue()	Read eFuse data using ReadFuseByte and ReadFuseByteMargin system calls, and compare these read data	–
CompareEfuseValue_DeadAccess()	Read Dead access restriction and compare with write value	–
CompareEfuseValue_SecureAccess()	Read Secure access restriction and compare with write value	–
Cy_ReadFuseByte()	Run the ReadFuseByte system call	–
Cy_ReadFuseByteMargin()	Run the ReadFuseByteMargin system call	–
Cy_TransitionToSecure ()	Run the TransitiontoSecure system call	–

The following code show an example of running TransitiontoSecure system call with use case:

Code listing 7 Sample software of TransitiontoSecure

```
#include "cy_project.h"
#include "cy_device_headers.h"
#include <stdio.h>

/** IMO Frequency **/
#define CLK_FREQ_IMO                    (8000000ul)
/** FLL Target Frequency **/
#define FLL_TARGET_FREQ                 (100000000ul)
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION          (10000ul)
/*** API OPCODE ***/
#define CY_SROM_OP_TRANSITION_TO_SECURE (0x2FUL)
/* API Parameters definition */
#define SECURE                          (0x0ul)
#define SECURE_ACC_RESTRICT             (0x00000000ul) /* For SECURE- It can be change by customer needs. */
#define DEAD_ACC_RESTRICT               (0x00000000ul) /* For SECURE */

/* API Parameters setting */
/* Nominal resistance (default read condition) */
uint32_t MarginCTL  = 1;
/*  0x01 and 0x2C - 0x43 */
uint32_t EfuseAddr[25] = {1,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67};
/* Secure Access Restrictions (0x3C-0x3E) */
uint32_t SecureAccessRestrictions_EfuseAddr[3] = {60,61,62};
/* Secure Dead Access Restrictions (0x40-0x42)  */
uint32_t DeadAccessRestrictions_EfuseAddr[3] = {64,65,66};

/* Global Variable */
uint32_t g_RdEfuseValue = 0;
uint32_t g_RdEfuseMarginValue50 = 0;
uint32_t g_RdEfuseMarginValue100 = 0;

void ClockSetTo100MHz(void);
```

```
bool Is_CLK100MHz(void);
bool CompareEfuseValue(uint32_t ReadEfuseAddr);
bool CompareEfuseValue_DeadAccess(uint32_t ReadEfuseAddr);
bool CompareEfuseValue_SecureAccess(uint32_t ReadEfuseAddr);
cy_en_srom_driver_status_t Cy_ReadFuseByte(uint32_t ReadEfuse_Efuseaddr);
cy_en_srom_driver_status_t Cy_ReadFuseByteMargin(uint32_t Margin_efuseAddr, uint32_t Margin_Ctl);
cy_en_srom_driver_status_t Cy_TransitionToSecure(uint32_t SecureDebug, uint32_t SecureAccessRestrict, uint32_t SecureDeadAccessRestrict);


int main(void)
{
    __enable_irq();

    /*** Initial clock setting for application ***/
    SystemInit();

    /*** Disable D-cache if need ***/

    /*** Set the CLK as FLL = 100MHz ***/
    ClockSetTo100MHz();
                                                                                 (2) See Code listing 5
    /*** Check CLK as expected ***/
    while(Is_CLK100MHz() != true);
                                                                                 (3) See Code listing 6
    /***  API Result ***/
    cy_en_srom_driver_status_t Result;

    /***  Call API TransitionToSecure     ***/
    Result = Cy_TransitionToSecure(SECURE,SECURE_ACC_RESTRICT,DEAD_ACC_RESTRICT);
    while(Result != CY_SROM_DR_SUCCEEDED);                                      (4) See Code listing 8

    int Num = 0;
    for(Num = 0;Num < 25;Num++)
    {
        /* Read byte adress from  0x01, 0x2C - 0x43 and compare the read value */
        if(CompareEfuseValue(EfuseAddr[Num]) != true)                            (5) See Code listing 9
        {
            /* mismatch*/                                                        (6)
            while(1);
        }
    }




    for(Num = 0;Num < 3;Num++)
    {
        /* Read byte adress from 0x3C - 0x3E, and compare the read value */      (7) See Code listing 10
        if(CompareEfuseValue_SecureAccess(SecureAccessRestrictions_EfuseAddr[Num]) != true)
        {
            /* mismatch*/                                                        (8)
            while(1);
        }
    }

    for(Num = 0;Num < 3;Num++)
    {
        /* Read byte adress from 0x40 - 0x42, and compare the read value */      (9) See Code listing 11
        if(CompareEfuseValue_DeadAccess(DeadAccessRestrictions_EfuseAddr[Num]) != true)
        {
            /* mismatch*/                                                        (10)
            while(1);
        }
    }

    for(;;);

}
```

Code listing 8 Cy_TransitionToSecure function

```
cy_en_srom_driver_status_t Cy_TransitionToSecure(uint32_t SecureDebug, uint32_t SecureAccessRestrict, uint32_t SecureDeadAccessRestrict)
{
    /* Prepares arguments to be passed to SROM API */
    un_srom_api_args_t apiArgs  = { 0ul };
    apiArgs.TransitionToSecure.arg0.debug            = SecureDebug;
    apiArgs.TransitionToSecure.arg0.opcode           = CY_SROM_OP_TRANSITION_TO_SECURE;
    apiArgs.TransitionToSecure.arg1.secureaccrest    = SecureAccessRestrict;
    apiArgs.TransitionToSecure.arg2.deadaccrest      = SecureDeadAccessRestrict;

    /* Call SROM API driver and process response */
    while(Cy_Srom_CallApi(&apiArgs, NULL) != CY_SROM_DR_SUCCEEDED);

    return CY_SROM_DR_SUCCEEDED;
}

cy_en_srom_driver_status_t Cy_ReadFuseByte(uint32_t ReadEfuse_Efuseaddr)
{
    /* Prepares arguments to be passed to SROM API */
    un_srom_api_args_t apiArgs  = { 0ul };
    un_srom_api_resps_t apiResp = { 0ul };
    apiArgs.RdFuse.arg0.eFuseAddr   = ReadEfuse_Efuseaddr;
    apiArgs.RdFuse.arg0.Opcode      = CY_SROM_OP_READ_FUSE_BYTE;

    /* Call SROM API driver and process response */
    while(Cy_Srom_CallApi(&apiArgs, &apiResp) != CY_SROM_DR_SUCCEEDED);

    g_RdEfuseValue = apiResp.RdFuse.resp0.ReadByte;

    return CY_SROM_DR_SUCCEEDED;
}
```

Code listing 9 CompareEfuseValue function

```
bool CompareEfuseValue(uint32_t ReadEfuseAddr)
{
    /***  API Result ***/
    cy_en_srom_driver_status_t Result;

    /***  Call API ReadFuseByte     ***/
    Result = Cy_ReadFuseByte(ReadEfuseAddr);                                          (5)-1
    while(Result != CY_SROM_DR_SUCCEEDED);

    /***  Call API ReadFuseByteMargin  MarginCTL = -50%     ***/                      (5)-2
    MarginCTL = 0; /* 0: Low resistance */
    Result = Cy_ReadFuseByteMargin(ReadEfuseAddr, MarginCTL);
    while(Result != CY_SROM_DR_SUCCEEDED);

    /***  Call API ReadFuseByteMargin  MarginCTL = +100%     ***/
    MarginCTL = 3; /* High resistance +100% */                                        (5)-3
    Result = Cy_ReadFuseByteMargin(ReadEfuseAddr, MarginCTL);
    while(Result != CY_SROM_DR_SUCCEEDED);
                                                                                      (5)-4
    if((g_RdEfuseValue == g_RdEfuseMarginValue50) && (g_RdEfuseValue == g_RdEfuseMarginValue100))
    {
        // Matched
        return true;                                                                  (5)-5
    }
    else
    {
        return false;
    }
}
```

Code listing 10 CompareEfuseValue_SecureAccess function

```
{
    /***  API Result ***/
    cy_en_srom_driver_status_t Result;

    /***  Call API ReadFuseByte     ***/
    Result = Cy_ReadFuseByte(ReadEfuseAddr);
    while(Result != CY_SROM_DR_SUCCEEDED);                                   (7)-1

    /* SECURE_ACC_RESTRICT */
    /* Compare the bit value of SECURE_ACC_RESTRICT */
    uint32_t SECURE_ACC_bitsValue = 0;
    if(ReadEfuseAddr == 60)
    {
        SECURE_ACC_bitsValue = (SECURE_ACC_RESTRICT & 0x000000FF);           (7)-2
        if(SECURE_ACC_bitsValue != g_RdEfuseValue)
        {                                                                    (7)-3
            /* Mismatched */
            return false;
        }
        else
        {
            /* Matched*/
            return true;
        }
    }
    else if(ReadEfuseAddr == 61)
    {
        SECURE_ACC_bitsValue = ((SECURE_ACC_RESTRICT & 0x0000FF00)>>8);
        if(SECURE_ACC_bitsValue != g_RdEfuseValue)
        {
            /* Mismatched */
            return false;
        }
        else
        {
            /* Matched*/
            return true;
        }
    }
    else if(ReadEfuseAddr == 62)
    {
        SECURE_ACC_bitsValue = ((SECURE_ACC_RESTRICT & 0x001F0000)>>16);
        if(SECURE_ACC_bitsValue != g_RdEfuseValue)
        {
            /* Mismatched */
            return false;
        }
        else
        {
            /* Matched*/
            return true;
        }
    }
    else
    {
        /* Please check parameter */
        return false;
    }
}
```

Code listing 11 CompareEfuseValue_DeadAccess function

```
{
    /***  API Result ***/
    cy_en_srom_driver_status_t Result;

    /***  Call API ReadFuseByte     ***/
    Result = Cy_ReadFuseByte(ReadEfuseAddr);
    while(Result != CY_SROM_DR_SUCCEEDED);                                  (7) -1

    /* DEAD_ACC_RESTRICT */
    /* Compare the bit value of DEAD_ACC_RESTRICT  */
    uint32_t DEAD_ACC_bitsValue = 0;
    if(ReadEfuseAddr == 64)
    {                                                                       (7) -2
        DEAD_ACC_bitsValue = (DEAD_ACC_RESTRICT & 0x000000FF);
        if(DEAD_ACC_bitsValue != g_RdEfuseValue)
        {
            /* Mismatched */                                                (7) -3
            return false;
        }
        else
        {                                                                   (7) -2
            /* Matched*/
            return true;         }
    }
    else if(ReadEfuseAddr == 65)                                            (7) -3
    {
        DEAD_ACC_bitsValue = ((DEAD_ACC_RESTRICT & 0x0000FF00)>>8);
        if(DEAD_ACC_bitsValue != g_RdEfuseValue)                            (7) -2
        {
            /* Mismatched */
            return false;                                                   (7) -3
        }
        else
        {
            /* Matched*/
            return true;
        }
    }
    else if(ReadEfuseAddr == 66)
    {
        DEAD_ACC_bitsValue = ((DEAD_ACC_RESTRICT & 0x001F0000)>>16);       (7) -2
        if(DEAD_ACC_bitsValue != g_RdEfuseValue)
        {
            /* Mismatched */
            return false;                                                   (7) -3
        }
        else
        {
            /* Matched*/
            return true;
        }
    }
    else
    {
        /* Please check parameter */
        return false;
    }
}
```

References

The following are the TRAVEO™ T2G family series datasheets and technical reference manuals. A sample driver library (SDL) including startup as sample software to access various peripherals is provided. The SDL also serves as a reference to customers, for drivers that are not covered by the official AUTOSAR products. The SDL cannot be used for production purposes as it does not qualify to any automotive standards. The code snippets in this application note are part of the SDL. To obtain the SDL and other documents, contact Infineon Technical Support.

[1] Device datasheet

- CYT2B6 datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ T2G family
- CYT2B7 datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ T2G family
- CYT2B9 datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ T2G family
- CYT2BL datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ T2G family
- CYT3BB/4BB datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family
- CYT4BF datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family
- CYT3DL datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family
- CYT4DN datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family

[2] Reference manual

- Body controller entry family
  - TRAVEO™ T2G Automotive Body Controller Entry family architecture technical reference manual (TRM)
  - TRAVEO™ T2G Automotive Body Controller Entry registers technical reference manual (TRM) for CYT2B6/CYT2B7
  - TRAVEO™ T2G Automotive Body Controller Entry registers technical reference manual (TRM) for CYT2B9
  - TRAVEO™ T2G Automotive Body Controller Entry registers technical reference manual (TRM) for CYT2BL (Doc No. 002-29852)
- Body controller high family
  - TRAVEO™ T2G Automotive Body Controller High family architecture technical reference manual (TRM)
  - TRAVEO™ T2G Automotive Body Controller High registers technical reference manual (TRM) for CYT4BF
  - TRAVEO™ T2G Automotive Body Controller High registers technical reference manual (TRM) for CYT3BB/4BB
- Cluster 2D family
  - TRAVEO™ T2G Automotive Cluster 2D family architecture technical reference manual (TRM) (Doc No. 002-25800)
  - TRAVEO™ T2G Automotive Cluster 2D registers technical reference manual (TRM) for CYT4DN (Doc No. 002-25923)
  - TRAVEO™ T2G Automotive Cluster 2D registers technical reference manual (TRM) for CYT3DL (Doc No. 002-29854)

[3] Application note

- AN228680 - Secure system configuration in TRAVEO™ T2G family

[4] Knowledge base article

- KBA234634 - Configure Software Protection Unit (SWPU) objects in TRAVEO™ T2G devices

Revision history

Document revision	Date	Description of changes
V 1.0	2023-12-07	Initial release
V 2.0	2025-01-22	Added detail description in Table 10 Fixed to correct link in 6 How to use system calls Added Note to 6.1.2 Software flow

Edition 2025-01-22

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

AN0002