TMS320C674x/OMAP-L1x Processor Security

User's Guide

Literature Number: SPRUGQ9
June 2011

Contents

List of Figures

List of Tables

Read This First

About This Manual

This document describes the security subsystem.

Notational Conventions

This document uses the following conventions.

- Hexadecimal numbers are shown with the suffix h. For example, the following number is 40 hexadecimal (decimal 64): 40h.
- Registers in this document are shown in figures and described in tables.
  - Each register figure shows a rectangle divided into fields that represent the fields of the register. Each field is labeled with its bit name, its beginning and ending bit numbers above, and its read/write properties below. A legend explains the notation used for the properties.
  - Reserved bits in a register figure designate a bit that is used for future device expansion.

Security Notice

TI cannot be held responsible in any way if security features implemented in or with TI solutions fail to prevent skilled hackers or thieves from gaining access to, copying, changing, deleting and/or damaging data, information, applications, code, functions, features, operation or security features associated with Customer's products. Customer assumes any such risk.

Recipient agrees that it will not knowingly export or re-export, directly or indirectly, any product or technical data (as defined by the U.S, EU and other Export Administration Regulations) including software, or any controlled product restricted by other applicable national regulations, received from Disclosing party under this Agreement, or any direct product of such technology, to any destination to which such export or re-export is restricted or prohibited by U.S or other applicable laws, without obtaining prior authorization from U.S. Department of Commerce and other competent Government authorities to the extent required by those laws. This provision shall survive termination or expiration of this Agreement.

According to our best knowledge of the state and end-use of this product or technology, and in compliance with the export control regulations of dual-use goods in force in the origin and exporting countries, this technology is classified as follows:

- US ECCN: 5E992
- EU ECCN: 5E992
- Other country ECCN: 5E992

and may require export or re-export license for shipping it in compliance with certain countries regulations.

Related Documentation From Texas Instruments

The following documents describe the TMS320C674x Digital Signal Processors (DSPs) and OMAP-L1x Applications Processors. Copies of these documents are available on the Internet at www.ti.com. *Tip:* Enter the literature number in the search box provided at www.ti.com.

The current documentation that describes the DSP, related peripherals, and other technical collateral, is available in the C6000 DSP product folder at: www.ti.com/c6000.

SPRUGM5 — ***TMS320C6742 DSP System Reference Guide.*** Describes the C6742 DSP subsystem, system memory, device clocking, phase-locked loop controller (PLLC), power and sleep controller (PSC), power management, and system configuration module.

SPRUGJ0 — ***TMS320C6743 DSP System Reference Guide.*** Describes the System-on-Chip (SoC) including the C6743 DSP subsystem, system memory, device clocking, phase-locked loop controller (PLLC), power and sleep controller (PSC), power management, and system configuration module.

SPRUFK4 — ***TMS320C6745/C6747 DSP System Reference Guide.*** Describes the System-on-Chip (SoC) including the C6745/C6747 DSP subsystem, system memory, device clocking, phase-locked loop controller (PLLC), power and sleep controller (PSC), power management, and system configuration module.

SPRUGM6 — ***TMS320C6746 DSP System Reference Guide.*** Describes the C6746 DSP subsystem, system memory, device clocking, phase-locked loop controller (PLLC), power and sleep controller (PSC), power management, and system configuration module.

SPRUGJ7 — ***TMS320C6748 DSP System Reference Guide.*** Describes the C6748 DSP subsystem, system memory, device clocking, phase-locked loop controller (PLLC), power and sleep controller (PSC), power management, and system configuration module.

SPRUG84 — ***OMAP-L137 Applications Processor System Reference Guide.*** Describes the System-on-Chip (SoC) including the ARM subsystem, DSP subsystem, system memory, device clocking, phase-locked loop controller (PLLC), power and sleep controller (PSC), power management, ARM interrupt controller (AINTC), and system configuration module.

SPRUGM7 — ***OMAP-L138 Applications Processor System Reference Guide.*** Describes the System-on-Chip (SoC) including the ARM subsystem, DSP subsystem, system memory, device clocking, phase-locked loop controller (PLLC), power and sleep controller (PSC), power management, ARM interrupt controller (AINTC), and system configuration module.

SPRUFK9 — ***TMS320C674x/OMAP-L1x Processor Peripherals Overview Reference Guide.*** Provides an overview and briefly describes the peripherals available on the TMS320C674x Digital Signal Processors (DSPs) and OMAP-L1x Applications Processors.

SPRUFK5 — ***TMS320C674x DSP Megamodule Reference Guide.*** Describes the TMS320C674x digital signal processor (DSP) megamodule. Included is a discussion on the internal direct memory access (IDMA) controller, the interrupt controller, the power-down controller, memory protection, bandwidth management, and the memory and cache.

SPRUFE8 — ***TMS320C674x DSP CPU and Instruction Set Reference Guide.*** Describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C674x digital signal processors (DSPs). The C674x DSP is an enhancement of the C64x+ and C67x+ DSPs with added functionality and an expanded instruction set.

SPRUG82 — ***TMS320C674x DSP Cache User's Guide.*** Explains the fundamentals of memory caches and describes how the two-level cache-based internal memory architecture in the TMS320C674x digital signal processor (DSP) can be efficiently used in DSP applications. Shows how to maintain coherence with external memory, how to use DMA to reduce memory latencies, and how to optimize your code to improve cache efficiency. The internal memory architecture in the C674x DSP is organized in a two-level hierarchy consisting of a dedicated program cache (L1P) and a dedicated data cache (L1D) on the first level. Accesses by the CPU to the these first level caches can complete without CPU pipeline stalls. If the data requested by the CPU is not contained in cache, it is fetched from the next lower memory level, L2 or external memory.

Security

1 Introduction

This section provides an overview of the security concepts implemented on basic secure boot devices.

1.1 Overview

The secure environment is supported by hardware within the C674x DSP core. This allows critical code to be executed in a secure environment, hiding sensitive information from the outside world with the help of the following hardware features:

- Secure thread support – hardware support that allows trusted code to be executed from secure ROM or secure RAM.
- Protected transitions between the secure and non-secure world via Secure Kernel software provided in the secure ROM.
- Ability to secure portions of internal and external RAM. Note that external RAM is only secure from a software point of view and is still vulnerable to physical device attacks if its contents are not encrypted.
- Secure ROM, containing secure boot, cryptographic, and Secure Kernel support.

1.2 Device Types

The production configuration of the TMS320C6748/OMAP-L138 devices is defined by 8 eFuse bits, and is therefore not modifiable in the field.

Basic secure boot devices are intended to only provide boot-time protection of a customer's code. All or part of the code and data stored in a customer's boot image can be encrypted. Furthermore, the boot image is tied to an individual device, such that it cannot be used to boot another device. This provides IP protection for the customer's product, at the expense of a more secure and rigorous production environment.

1.3 Boot-time Security

Basic secure DSP boot is the foundation of security on the device. It allows the boot image to be validated and authenticated before use, ensuring that the customer software has not been modified from the original customer signed image. Basic secure boot devices always boot from the DSP internal secure ROM, enforcing a secure boot mechanism where code loaded on the platform is authenticated before it is executed. Only valid boot images, with the proper format, encryption, keys, and signatures, are accepted.

On basic secure boot devices, the loadable code and data sections of the boot image can be signed only, or signed and also encrypted using the customer encryption key (CEK), which is loaded as part of the boot image.

1.4 ROM Boot Loader

The ROM boot loader resides in the ROM of the device beginning at global address 1170 0000h (local DSP address 0070 0000h). The ROM boot loader (RBL) implements methods for booting various boot modes. It reads the contents of the BOOTCFG register to determine boot mode and performs appropriate commands to effect boot of the device. If an improper boot mode is chosen or if for some reason an error is detected during boot, the RBL sets an error condition and then goes to an abort loop.

When booting in master mode, the boot loader reads the boot information from the slave device as and when required. When booting in slave mode, the boot loader depends on the external master device to feed the boot information as and when required. The reader should consult the non-secure boot loader documentation for further details of the supported boot modes and the basics of the AIS format.

1.5 Terminology Used in this Document

Term	Definition
AES	Advanced Encryption Standard – An encryption process that employs symmetric encryption, using a single key to both encrypt and decrypt a message.
BIST	Built-In Self Test
DRM	Digital Rights Management – A system for protecting the copyrights of data circulated in electronic form by enabling secure distribution and/or disabling unauthorized distribution of the data.
DSP	Digital Signal Processor – A specialized processor designed specifically for digital signal processing.
DSP/BIOS	A scalable real-time kernel, designed specifically for the TMS320C5000 and TMS320C6000 DSP platforms.
FIPS	Federal Information Processing Standards – United States Government technical standards published by the National Institute of Standards and Technology (NIST).
GEM	Generalized Embedded Megamodule – TI name for the C674x DSP core.
ICEPick	ICEPick In-Circuit Emulation TAP Selection Module.
ISTP	Internal Service Table Pointer – A register in the C674x DSP architecture that is used to locate interrupt service routines.
JTAG	Joint Test Action Group – A standard for boundary scan controllers, specifying how to control and monitor the pins of compliant devices on a board. Also referred to as IEEE Standard 1149.1.
LBIST	Logic Built-In Self Test
LED	Load/Execute/Dump – A production test sequence
MBIST	Memory Built-In Self Test
MMR	Memory-Mapped Register
NAND flash	A type of nonvolatile memory in which the cells are linked serially to each other and the grid of interconnects. NAND is best suited for sequential data reads and writes.
NIST	National Institute of Standards and Technology – A United States governmental body that helps develop standards including FIPS.
NOR flash	A fast type of nonvolatile memory in which the cells are linked in parallel to the grid. NOR is suited for random-access usage.
OS	Operating System
PBIST	Power-on Built-In Self Test
PKCS	Public Key Cryptography Standard – A group of standards created and published by RSA Laboratories, the research center of RSA Security Inc.
POR	Power-on Reset
RAM	Random Access Memory
REP	Restricted Entry Point – A register in the C674x DSP architecture that is used to designate the entry point of the internal exception handler used to request Supervisor / Secure mode services.
ROM	Read Only Memory – A type of data storage device which is manufactured with fixed contents.
RPC	Remote Procedure Call – A protocol which allows a program running on one processor to cause code to be executed on another processor without the programmer needing to explicitly code for this.
SHA-1	Secure Hash Algorithm – A one-way hash function developed by NIST.
TAP	Test Access Port

2 Hardware Security Features

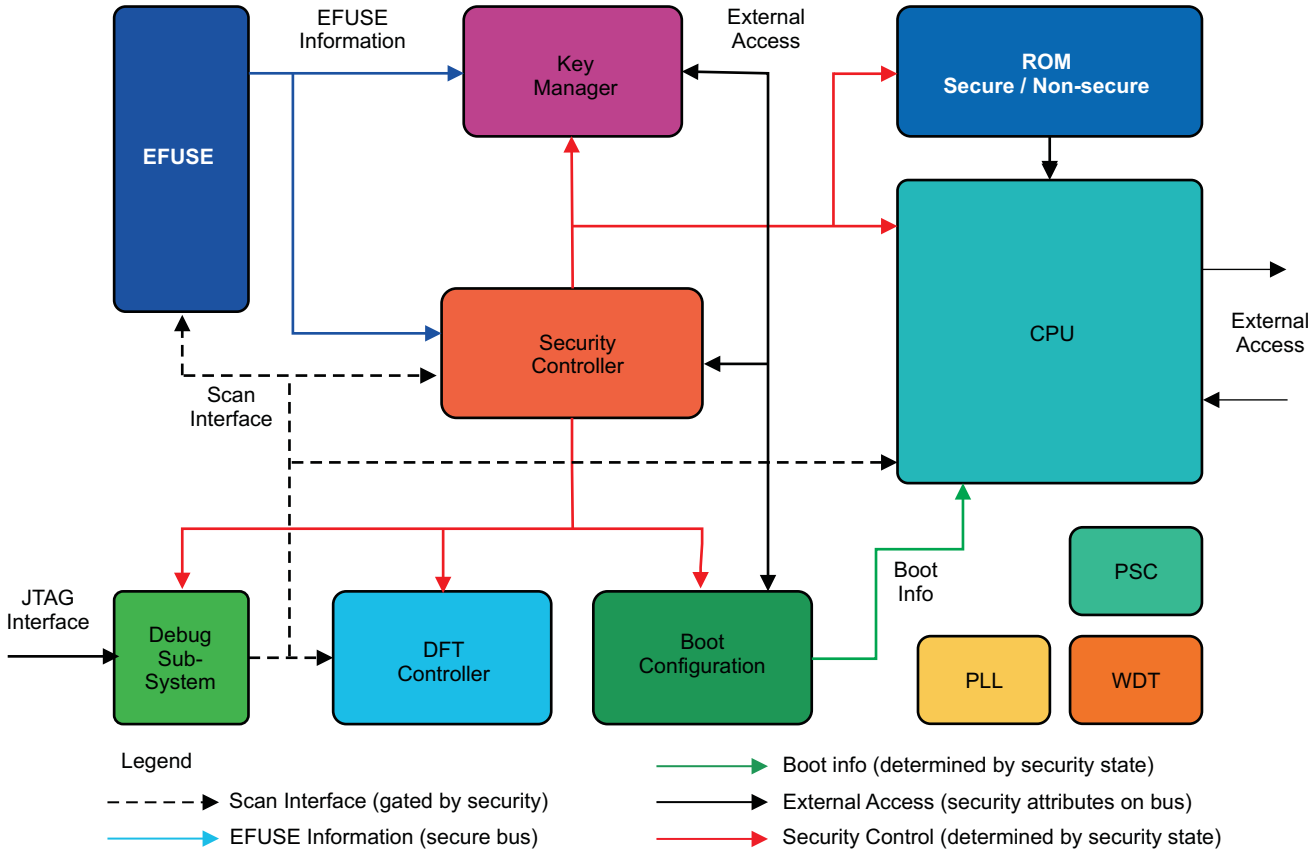
This section provides an overview of the hardware security features.

2.1 Security Hardware Overview

The devices offers a secure variant that is described throughout this document. There are a few different use cases and operating modes for a secure boot device. There are also a number of different hardware components of the SoC that enable security features. The core chip components that control security features are shown in Figure 1.

The CPU block in Figure 1 is the C674x DSP. On OMAP-L1x devices, the ARM9 processor of the device is not part of the security system and always runs as a non-secure master in the system.

Figure 1. Security Hardware Overview



2.2 Secure Mode

On the basic secure boot devices, all C674x DSP security features are enabled upon power up. Upon reset, the C674x core starts executing from Secure ROM in Secure Supervisor mode. As a result, the device is secured from the initial power-up sequence and any changes in the secure state of the device must be made by or through the Secure ROM bootloader or an application securely loaded via the Secure ROM boot loader.

2.3 Security versus Privilege

A distinction is made between security levels versus privilege levels in basic secure boot devices. The most important point to be understood is that security is orthogonal to privilege. The privilege levels of the device are “Supervisor” or “User”. The security levels of the device are “Secure” or “Non-secure”. Table 1 shows the orthogonality in graphical form.

Internal bus transactions contain information about the security and privilege level of the master that initiated them. Bus slaves use this information to determine how to react to the master’s requests.

The roles of privilege in the system include the following:

- Protects relatively fixed OS from more dynamic application-level code.
- Insulates stable/proven code from non-trusted newer/unproven code.
- Allows finer-grain recovery from transient application failures.

In contrast, the roles of security in the system include the following:

- Protects vendors from customers, competing vendors and pirates.
- Protects consumers from information and identity theft.
- Enforces usage restrictions set by vendors, despite allowing great programmability for the device.

Table 1. Security versus Privilege

	Security	
Privilege	**Secure**	**Non-Secure**
Supervisor	Secure Kernel and Secure Boot Loader	DSP/BIOS or other OS Kernels
User	Licensed Algorithms (for example, WMV, WMA, etc.)	Non-Secure Applications on other OS Kernels

2.4 Security Keys

A basic secure boot device supports one 128-bit eFuse programmable security key:

- Key Encryption Key (KEK) – a random 128-bit key, unique to the device, without any record of the key value (unknown outside of the device itself)

This key is internally provided to the system via the key manager block. The keys and other registers of the key manager are only visible in Secure Supervisor mode; they will become invisible if the device is switched to a non-secure operating mode. In the case of these basic secure boot devices, security is maintained through the random KEK, which remains unknown outside of the chip.

2.5 Hardware Security Controller Registers

As shown in Figure 1, the system security controller is central to the hardware security mechanisms of basic secure boot devices. The security control interface consists of a block of seven 32-bit registers, as listed in Table 2.

Table 2. Security Controller Registers

Offset	Acronym	Register Description	Section
3000h	PID	Security controller peripheral ID register	Section 2.5.1
3010h	SYSSTATUS	System security status register	Section 2.5.2
3014h	SYSWR	System security write-once register	Section 2.5.3
3020h	SYSCONTROL	System security control register (multiple read and write by any master)	Section 2.5.4
3024h	SYSCONTROLPROTECT	System security protected control register (multiple read and write by only secure supervisor masters)	Section 2.5.5
3028h	SYSTAPEN	System TAP enable register	Section 2.5.6
302Ch	SECRESERVED	Security reserved register	Section 2.5.7

2.5.1 Security Controller Peripheral ID Register (PID)

The security controller peripheral ID register (PID) is shown in Figure 2 and described in Table 3.

Figure 2. Security Controller Peripheral ID Register (PID)

31	30	29	28	27	16
SCHEME		Reserved		FUNC	
R-1		R-0		R-484h	

15	11	10	8	7	6	5	0
RTL		MAJOR		CUSTOM		MINOR	
R-x		R-2h		R-0		R-0	

LEGEND: R = Read only; -*n* = value after reset; -x = value is indeterminate after reset

Table 3. Security Controller Peripheral ID Register (PID) Field Descriptions

Bit	Field	Value	Description
31-30	SCHEME	1	Always read as 01.
29-28	Reserved	0	Reserved. Always read as 0.
27-16	FUNC	0-FFFh	Functional group
15-11	RTL	0-1Fh	RTL revision
10-8	MAJOR	0-7h	Major revision
7-6	CUSTOM	0-3h	Custom field
5-0	MINOR	0-3Fh	Minor revision

2.5.2 System Security Status Register (SYSSTATUS)

The system status register (SYSSTATUS) contains status information on the state of the security system. This is the only register that must be read to determine the status of the entire system. It is a read-only register and some bits are replicated in other registers. The SYSSTATUS is shown in Figure 3 and described in Table 4.

Figure 3. System Security Status Register (SYSSTATUS)

31	30	29	28	27	26	25	24
SECURE_EN_CTL							
R-0							

23	22	21	20	19	18	17	16
Reserved				TAP_EN_REG_WRITTEN	Reserved		SYSTEM_REG_WRITTEN
R-0				R-0	R-0		R-0

15	14	13	12	11	10	9	8
Reserved				SEC_ROM_PASS	SEC_ROM_DONE	CHKSUM_PASS	CHKSUM_DONE
R-0				R-0	R-0	R-0	R-0

7	6	5	4	3	2	1	0
MMR_UNLOCK	MMR_LOCK	SEC_OVERRIDE	TEST_AS_SEC	TEST_AS_NONSEC	DEVICE_TYPE		
R-0	R-0	R-0	R-0	R-0	R-2h		

LEGEND: R = Read only; -*n* = value after reset

Table 4. System Security Status Register (SYSSTATUS) Field Descriptions

Bit	Field	Value	Description
31-24	SECURE_EN_CTL	x	Secure enable control. Replicated bit field, see Section 2.5.5.
23-20	Reserved	0	Reserved. Always read as 0. Writes have no effect.
19	TAP_EN_REG_WRITTEN		System TAP enable register written. Indicates whether the system TAP enable register (SYSTAPEN) has been written. If written, no changes can be made until the next power-on reset (POR).
		0	Register is not yet written.
		1	Register is written; no further writes allowed.
18-17	Reserved	0	Reserved. Always read as 0. Writes have no effect.
16	SYSTEM_REG_WRITTEN		System security write-once register written. Indicates whether the system security write-once register (SYSWR) has been written. If written, no changes can be made until the next power-on reset (POR).
		0	Register is not yet written.
		1	Register is written; no further writes allowed.
15-12	Reserved	0	Reserved. Always read as 0. Writes have no effect.
11	SEC_ROM_PASS	x	Secure ROM pass. Replicated bit field, see Section 2.5.3.
10	SEC_ROM_DONE	x	Secure ROM done. Replicated bit field, see Section 2.5.3.
9	CHKSUM_PASS		Security key checksum pass. Indicates if the security keys passed their checksum calculations. This value is only valid when CHKSUM_DONE = 1. This calculation reflects the calculation on all keys within the device.
		0	Mismatch between key and checksum values.
		1	Match between key and checksum values.
8	CHKSUM_DONE		Security key checksum done. Indicates that the security keys checksum calculations are done and the pass/fail status can be read.
		0	Checksum calculations are not done.
		1	Checksum calculations are done.
7	MMR_UNLOCK	x	MMR unlock. Replicated bit field, see Section 2.5.5.

Table 4. System Security Status Register (SYSSTATUS) Field Descriptions (continued)

Bit	Field	Value	Description
6	MMR_LOCK	x	MMR lock. Replicated bit field, see Section 2.5.5.
5	SEC_OVERRIDE	x	Security override. Replicated bit field, see Section 2.5.5.
4	TEST_AS_SEC	0-1	Test as secure JTAG request. Indicates either the test_as_sec_mmr_req or the JTAG test_as_sec_jtag_req is asserted.
3	TEST_AS_NONSEC	x	Test as non-secure JTAG request. Replicated bit field, see Section 2.5.3.
2-0	DEVICE_TYPE		Device type. Indicates the device security type value.
		0	TEST
		1h	BAD
		2h	SECURE
		3h	NONSECURE
		4h-7h	BAD

2.5.3 System Security Write-once Register (SYSWR)

The system write-once register (SYSWR) provides control of the security system bits that should only be changed once during each power-on reset (POR). These bits are only allowed to be written by an allowed secure supervisor. The SYSWR is shown in Figure 4 and described in Table 5.

Figure 4. System Security Write-once Register (SYSWR)

31	16
Reserved	
R-0	

15	10	9	8	7	1	0
Reserved		SEC_ROM_PASS	SEC_ROM_DONE	Reserved		SEC_OVERRIDE
R-0		R/W-0	R/W-0	R-0		R/W-0

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

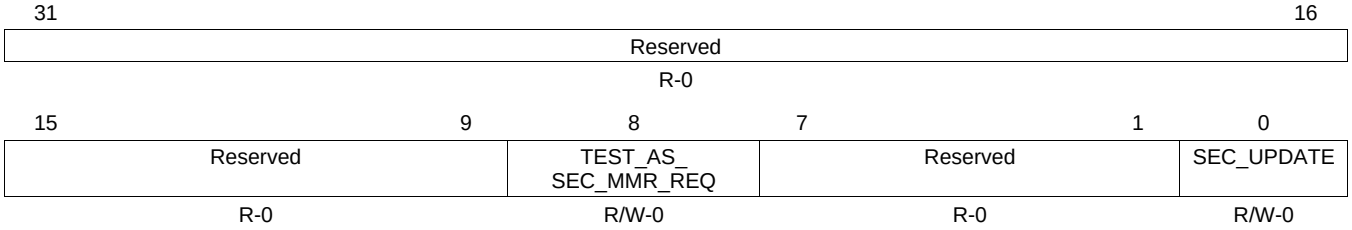
Table 5. System Security Write-once Register (SYSWR) Field Descriptions

Bit	Field	Value	Description
31-10	Reserved	0	Reserved. Always read as 0. Writes have no effect.
9	SEC_ROM_PASS		Secure ROM pass. Indicates whether the secure ROM passed all integrity checks during the override sequence (this includes checks like PBIST). This bit is only valid when SEC_ROM_DONE = 1.
		0	Secure ROM failed checks.
		1	Secure ROM passed checks.
8	SEC_ROM_DONE		Secure ROM done. Indicates whether the secure ROM contents have finished being validated as part of the override sequence.
		0	Secure ROM has not been validated.
		1	All checks have been performed on secure ROM.
7-1	Reserved	0	Reserved. Always read as 0. Writes have no effect.
0	SEC_OVERRIDE		Security override. Indicates whether a security override should take place.
		0	No security override.
		1	Security override granted.

2.5.4 System Security Control Register (SYSCONTROL)

The system control register (SYSCONTROL) provides control of the security system but is not as strict as other bits within the security system. These bits may be read or written by any master. The SYSCONTROL is shown in Figure 5 and described in Table 6.

Figure 5. System Security Control Register (SYSCONTROL)



LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 6. System Security Control Register (SYSCONTROL) Field Descriptions

Bit	Field	Value	Description
31-9	Reserved	0	Reserved. Always read as 0. Writes have no effect.
8	TEST_AS_ SEC_MMR_REQ	0-1	Test as secure MMR request. Value of 1 indicates that you are requesting to move to a higher security state within the device.
7-1	Reserved	0	Reserved. Always read as 0. Writes have no effect.
0	SEC_UPDATE		Security controller update. This bit causes the security controller to update its security outputs. Always read as 0.
		0	No effect.
		1	Causes the security controller to update its outputs.

2.5.5 System Security Protected Control Register (SYSCONTROLPROTECT)

The system control protected register (SYSCONTROLPROTECT) provides control of the security system. SYSCONTROLPROTECT should only be writeable by allowed non-debug, secure supervisor masters. These bits are important to the security system and must be protected but may need to be changed more than once during the course of device operation. The SYSCONTROLPROTECT is shown in Figure 6 and described in Table 7.

Figure 6. System Security Protected Control Register (SYSCONTROLPROTECT)

31	18	17	16
Reserved		MMR_UNLOCK	MMR_LOCK
R-0		R/W-0	R/W-0

15	8	7	0
Reserved		SECURE_EN_CTL	
R-0		R/W-0	

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 7. System Security Protected Control Register (SYSCONTROLPROTECT) Field Descriptions

Bit	Field	Value	Description
31-18	Reserved	0	Reserved. Always read as 0. Writes have no effect.
17	MMR_UNLOCK		MMR unlock. Indicates whether the system MMRs that are locked should be unlocked. The output of this bit is reflected on sec_mmr_init_lock_po.
		0	MMRs are locked.
		1	MMRs are unlocked.
16	MMR_LOCK		MMR lock. Indicates whether the system MMRs that are unlocked should be locked. The output of this bit is reflected on sec_mmr_init_unlock_po.
		0	MMRs are unlocked.
		1	MMRs are locked.
15-8	Reserved	0	Reserved. Always read as 0. Writes have no effect.
7-0	SECURE_EN_CTL		Secure enable control. Controls the value of the output port sec_secure_en_po[7:0]. These bits are used to enable security for non-security aware masters within the system.
		0	Master is not secure.
		1	Master is secure.

2.5.6 System TAP Enable Register (SYSTAPEN)

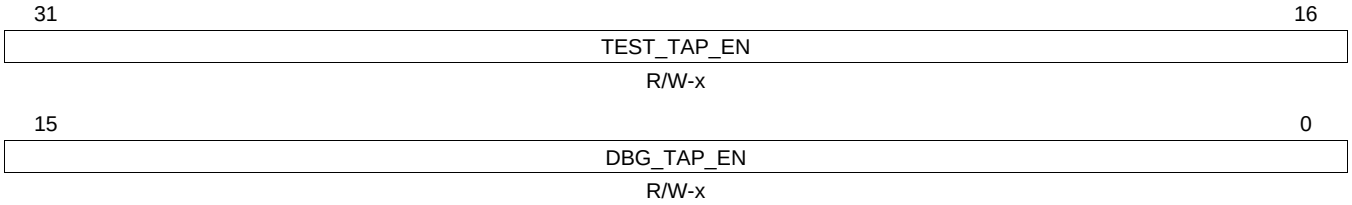
The system test access port (TAP) enable register (SYSTAPEN) enables or disables access to various modules that can be included on the JTAG scan chain. The SYSTAPEN is shown in Figure 7 and described in Table 8.

Each bit in SYSTAPEN drives a separate xSTAPEN output signal to ICEPick. Reset values for this register get latched when the device type is known. The reset values of SYSTAPEN for each of the device types are:

Device Type	SYSTAPEN Default Value
NON-SECURE	FFFF FFFFh
SECURE	FFFF 0000h

The bits of SYSTAPEN are write-once, and their value can be changed during the boot loading process to either open or close the JTAG access until the next power-on reset (POR). See Section 3 for more details.

Figure 7. System TAP Enable Register (SYSTAPEN)



LEGEND: R/W = Read/Write; -*n* = value after reset; -x = value is determined by device type

Table 8. System TAP Enable Register (SYSTAPEN) Field Descriptions

Bit	Field	Value	Description
31-16	TEST_TAP_EN		Test TAP enable. Reflects the value driven on the sec_test_tapenable_po signal that is routed to ICEPick.
		0	Test TAP is not selectable by ICEPick.
		1	Test TAP is selected by ICEPick.
15-0	DBG_TAP_EN		Debug TAP enable. Reflects the value driven on the sec_dbg_tapenable_po signal that is routed to ICEPick.
		0	Debug TAP is not selectable by ICEPick.
		1	Debug TAP is selected by ICEPick.

2.5.7 Security Reserved Register (SECRESERVED)

The security reserved register (SECRESERVED) contains status information stored in EFUSE that is reserved for future use. The SECRESERVED is shown in Figure 8 and described in Table 9.

Figure 8. Security Reserved Register (SECRESERVED)

31	24	23	0
Reserved		SEC_RSVD	
R-0		R-0	

LEGEND: R = Read only; -*n* = value after reset

Table 9. System TAP Enable Register (SYSTAPEN) Field Descriptions

Bit	Field	Value	Description
31-24	Reserved	0	Reserved. Always read as 0. Writes have no effect.
23-0	SEC_RSVD	0-FF FFFFh	Security reserved field. Security bits contained in the EFUSE that are reserved for security purposes. Currently, they are not defined within the security architecture but are available for future use.

3 Secure ROM Boot Loader

This section describes the various secure functionality of the ROM boot loader.

The boot sequence is the process by which the device's memory is loaded with program and data sections, and by which some of the device's initial setup and configuration can be performed. The boot sequence is started automatically after each device-level global reset.

This section details differences between the supported boot modes of the non-secure version and the basic secure boot version of the same device. Unless otherwise noted in this section, the boot modes and features of the non-secure boot loader apply to the secure boot loader. It is assumed that the reader is familiar with the contents of the non-secure boot loader application note for the device in use. See *Using the OMAP-L1x8 Bootloader Application Report* (SPRAB41) or *Using the TMS320C6748/C6746/C6742 Bootloader Application Report* (SPRAAT2).

3.1 Unsupported and Altered Boot Modes

The following boot modes are not supported on basic secure boot devices:

- NOR direct boot – Allowing unvalidated code to run from the EMIFA poses a security hole and is therefore prohibited.

The following boot modes have altered behavior on basic secure boot devices:

- NOR legacy boot – On non-secure devices, the NOR legacy boot copies data from the start of the EMIF region into the DSP L2 RAM and then executes it from that location. On basic secure boot devices, the data is similarly copied, but it must be validated, and possibly decrypted, before execution is allowed. To accomplish this, the boot image copied from the EMIF must be wrapped in a special format, including headers and footers. See Section 3.3.1 for details.
- UHPI boot – On non-secure devices, the UHPI boot mode allows an external host to place data into the L2 RAM and then indicate to the ROM that it should execute that code. Like the NOR legacy boot, validation and or decryption must take place before this can be allowed on a basic secure boot device. The image placed in the L2 RAM by the external host must be wrapped in the same manner previously mentioned for NOR legacy boot. See Section 3.3.2 for details.
- All AIS boot modes – On non-secure devices, most boot modes use the AIS format. This format must be altered and extended in some ways for basic secure boot devices to facilitate loading of keys, signature validation, and possibly decryption. The required changes to the AIS format are described in Section 3.2.

3.2 Extensions and Alterations to AIS: Secure AIS

The AIS format for non-secure boot images is thoroughly described in the documentation for the non-secure devices (see *Using the OMAP-L1x8 Bootloader Application Report* (SPRAB41) or *Using the TMS320C6748/C6746/C6742 Bootloader Application Report* (SPRAAT2). This section describes the changes, additions, and deletions for the secure version of the AIS format. All AIS information found in the non-secure boot loader documentation applies to the secure version of the device, unless it is specifically delineated as being different within this section.

3.2.1 Disallowed AIS Commands

On basic secure boot devices, there are certain AIS commands that cannot be used since they would pose a security risk.

This sections details the commands that are not allowed on basic secure boot devices. The reason these AIS commands are not allowed is that they would expose the secret Key Encryption Key (KEK) to malicious attackers.

- JUMP: The JUMP command (5853 5905h) is used to enter and execute user loaded code at boot time. On basic secure boot devices, where there is a possibility to boot the device using your own unencrypted key, an attacker could easily load code to read the KEK of a particular device, since the ROM boot loader executes in the Secure Supervisor context. Then the attacker could use that KEK to decrypt the key header of a protected image that had previously been tied to that device for boot purposes. This would reveal the true encryption key, allowing the attacker to then decrypt the remainder of the boot image and see the code it contained. To prevent this attack, this command is not allowed on basic secure boot devices.
- READ WAIT: The Read Wait command (5853 5914h) is used to loop reading a certain memory location, waiting until certain conditions are met within the returned value. This mechanism could be used to read the locations of the KEK and determine what its contents are. Therefore this command is not allowed on basic secure boot devices.
- FINAL FUNCTION REGISTER: The FINAL FUNCTION REGISTER command (5853 5915h) is used to register a certain loaded function for execution after the entire boot image has been read. This can be used to execute some final device configuration that could not have been executed during the retrieval of the actual boot image (because it would have altered the boot peripheral configuration, for example). This command opens the same hole as the JUMP command and is disallowed for the same reason.
- REQUEST CRC: Since the integrity of secure images is guaranteed using secure signatures, CRC checking is not required. Therefore the REQUEST CRC AIS command (5853 5902h) is not allowed.
- ENABLE CRC: Since the integrity of secure images is guaranteed using secure signatures, CRC checking is not required. Therefore the ENABLE CRC AIS command (5853 5903h) is not allowed.
- DISABLE CRC: Since the integrity of secure images is guaranteed using secure signatures, CRC checking is not required. Therefore the DISABLE CRC AIS command (5853 5904h) is not allowed.

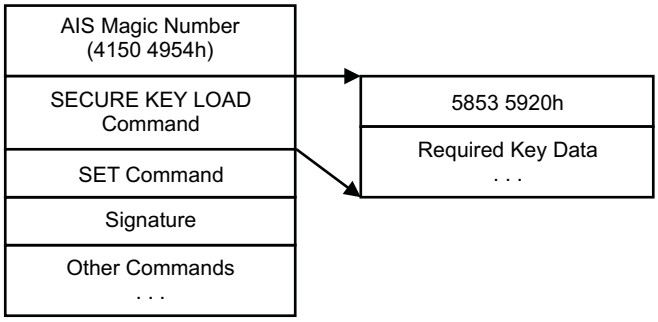
3.2.2 New AIS Commands

To support secure booting on basic secure boot devices, a number of extensions have been added to the standard AIS instruction set. This section details these commands and their proper usage.

3.2.2.1 SECURE KEY LOAD Command

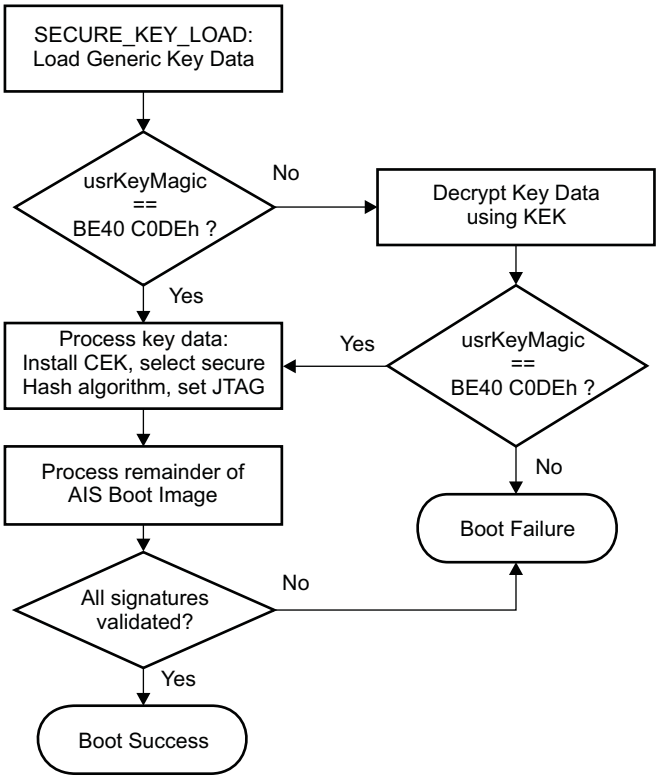
The SECURE KEY LOAD command (5853 5920h) must be the first AIS command immediately following the AIS magic number. If any other AIS command is attempted before the SECURE KEY LOAD command, the boot aborts. The format of the data delivered with the SECURE KEY LOAD command is shown in Figure 9.

Figure 9. Structure of SECURE KEY LOAD Command



The device has the capability to encrypt a key structure containing the true customer encryption key (CEK) using the device-specific KEK value. The SECURE KEY LOAD command loads this customer encryption key so it can be used to decrypt the data for subsequent ENCRYPTED SECTION LOAD commands. The flow of how the key data is installed is shown in Figure 10.

Figure 10. Boot Loader Flow for Basic Secure Boot Devices Showing Key Data Loading



The layout of the user key structure (after decryption) is given in Table 10. The hashAlgSelect field allows selection of what secure hash algorithm is used in the signatures for the basic secure boot image. The values are selected from those listed in Table 11.

Table 10. Structure of Key Data for Basic Secure Boot Devices

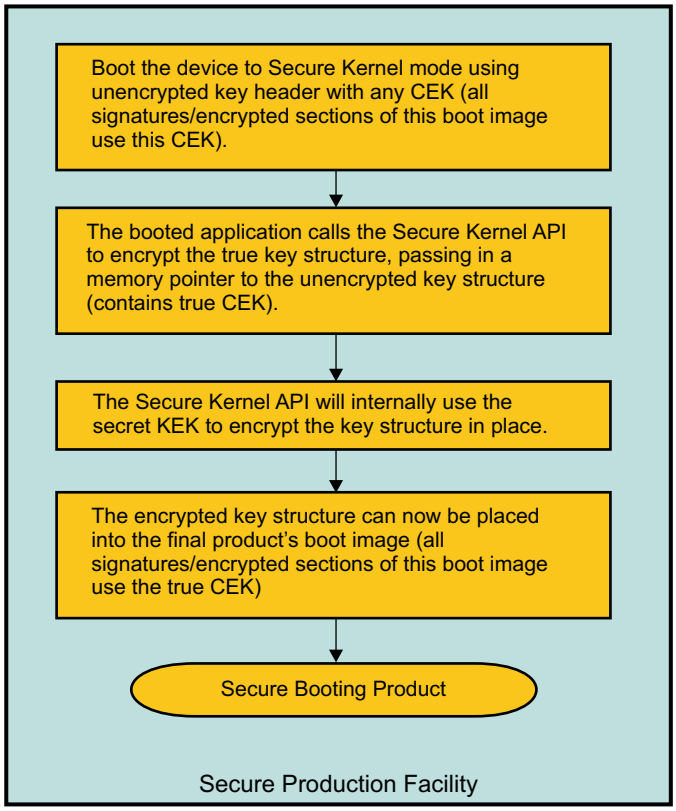
	Size		
Name	**Bits**	**Bytes**	**Description**
usrKeyMagic	32	4	Basic Key Data Magic Number = BE40 C0DEh
JTAGforceOff	32	4	Boolean to force JTAG permanently off
hashAlgSelect	32	4	Hash Algorithm selection value
randomSeed	32	4	Random Seed
CEK	128	16	Customer Encryption Key (AES-128)

Table 11. Hash Algorithm Values for hashAlgSelect Field

Hash Algorithm	Value
SHA-1	0
SHA-256	1h
SHA-384	2h (not supported)
SHA-512	3h (not supported)

The key data attached to the SECURE KEY LOAD command on basic secure boot devices does not have to be encrypted. In fact, in order to encrypt this structure with the device KEK, you must first boot the device in a mode with the Secure Kernel active, using an unencrypted header (see Figure 11). In this mode, you will be able to use the Secure Kernel interface to call into the secure ROM of the device to request that it encrypt the key structure without ever exposing the KEK (see the description of the SK_setUserKey API). The encrypted version of the key structure that is returned should be used when the product is actually deployed, thus hiding the true CEK. Since the key structure was encrypted using the chip's unique and random KEK, this structure is tied to the device upon which it was encrypted. Using this encrypted key structure in your final boot image, links the entire boot image to that particular chip. As long as the KEK of the device remains secret, the encrypted key structure cannot be decrypted, and the CEK within that structure is protected.

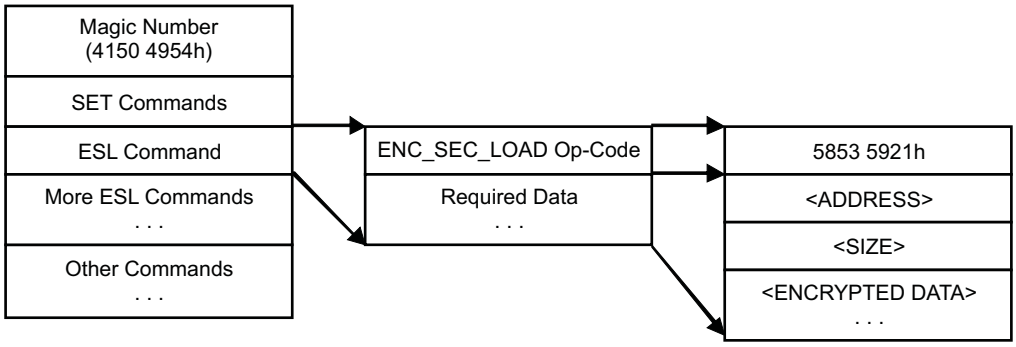
Figure 11. Sequence for Preparing Basic Secure Boot Images



3.2.2.2 *ENCRYPTED SECTION LOAD Command*

The ENCRYPTED SECTION LOAD command (5853 5921h) provides a way to load encrypted data sections (either application data or code). The encryption scheme used is AES CBC mode encryption with Cipher Text Stealing. The structure of the command is shown in Figure 12.

Figure 12. Structure of ENCRYPTED SECTION LOAD Command



Sections smaller than one AES block size (16 bytes) should be padded to the length of one AES block size and then encrypted using AES CBC only. In this particular case, the size parameter of the ENCRYPTED SECTION LOAD command should still reflect the actual size of the original section before padding. The ROM boot loader will know that it must read 16 bytes to properly perform the decryption.

Sections larger than one AES block size will use the normal Cipher Text Stealing algorithm. The Cipher Text Stealing algorithm allows the original section size and the encrypted section size to match exactly, with no need for padding. Therefore, the section size parameter in the ENCRYPTED SECTION LOAD command should reflect the true section size. The Initial Vector (IV) for all encrypted section AES CBC operations is the CEK AES encrypted with the CEK.

The cipher text stealing algorithm used for encrypted sections follows the NIST algorithm proposal CBC-CS (see *Proposal To Extend CBC Mode By* "*Ciphertext Stealing*", May 6, 2007).

The data is encrypted using the following algorithm shown in Figure 13. The $CIPH_K$ operation is AES-128 encryption. The actual data transferred corresponds to $C_1 \| C_2 \| ... \| C_n \| C_{-1}$* (that is, the partial block is transferred at the end of the sequence, so that the start of each block is aligned on AES block size boundaries).

The decoding algorithm is shown in Figure 14. The $CIPH^{-1}{}_K$ operation is AES-128 decryption.

In both diagrams, the stolen cipher text is denoted as C_{-1}**.

The receiver holds the partial block C_{-1}* and the first decrypts block C_N, resulting in a combination of P_N* XORed with C_{-1}*, and C_{-1}**, the stolen cipher text. Next, the stolen cipher text is combined with the held block C_{-1}*, resulting in the original cipher text block C_{-1}. This block can now be decrypted to yield plain text P_{-1}.

Finally, the last (partial) plain text block is reconstructed by XORing the partial bytes of the decrypted C_N block with C_{-1}*, yielding P_N*.

Figure 13. Encryption Flow Using AES CBC with Cipher Text Stealing

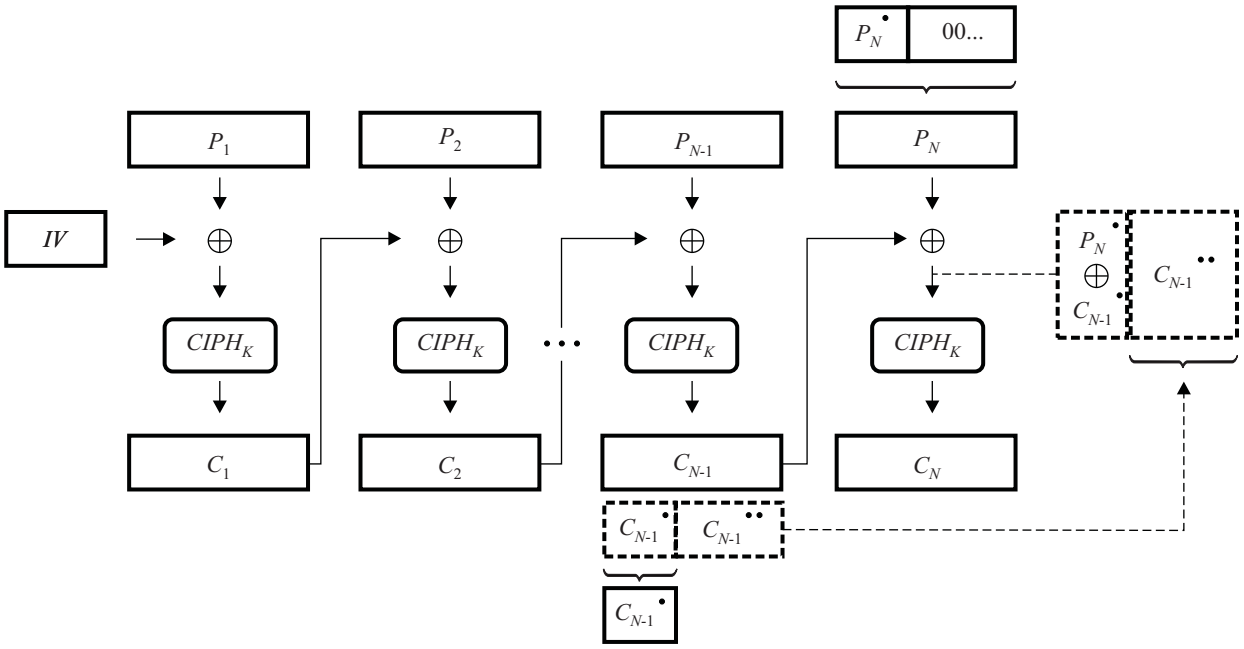
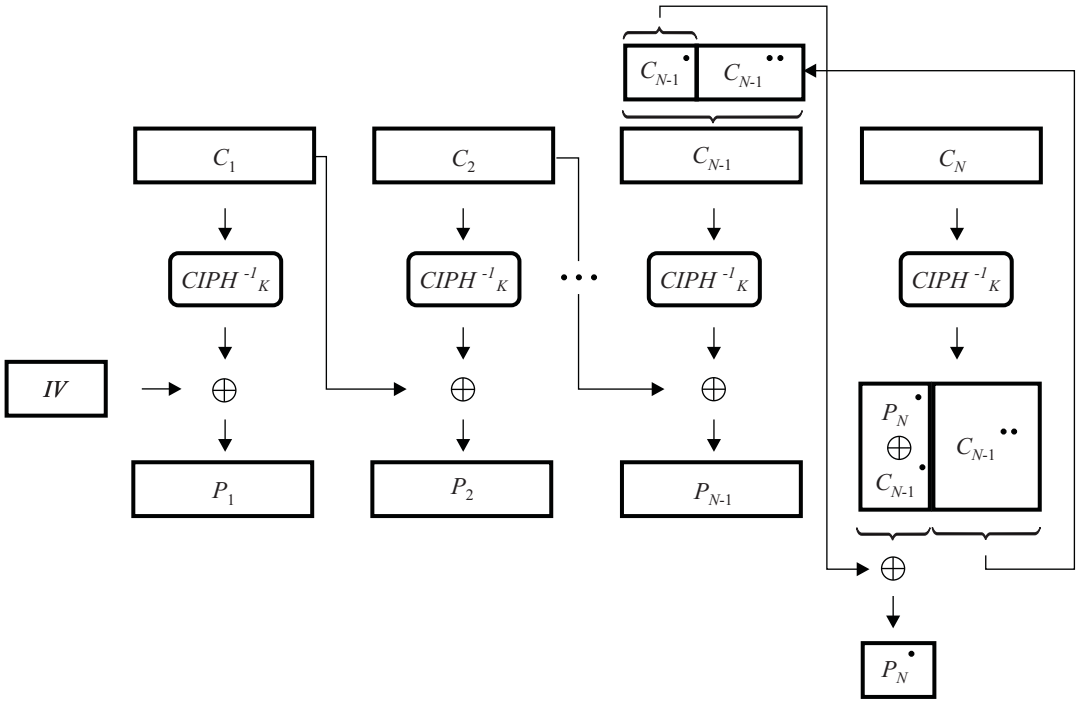


Figure 14. Decryption Flow for AES CBC with Cipher Text Stealing



3.2.2.3 SET EXIT MODE Command

The SET EXIT MODE command (5853 5923h) provides a way to specify the exit mode of the secure ROM boot loader. The structure of the command is shown in Figure 15. The different exit modes (Table 12) are useful depending on the intended use case for the device. The default exit type, if none is ever provided via this command during the boot, is NONSECURE.

On basic secure boot devices, the SECURE_NO_SK exit type is prohibited since this would directly expose the device KEK. The primary mode of operation for a basic secure boot device is the NONSECURE exit type. The basic secure boot device is intended to provide only boot-time security; therefore, switching the device to the non-secure operating state at the conclusion of boot is logical.

The only circumstance where the SECURE_WITH_SK option is useful is when the key header data needs to be encrypted so that it can be used in future secure boots. The Secure Kernel provides an API to do this (see SK_setUserKey and so an option to boot with the secure kernel active must be provided. The other APIs of the Secure Kernel are not very useful as there is no RSA public key installed.

Figure 15. Structure of SET EXIT MODE Command

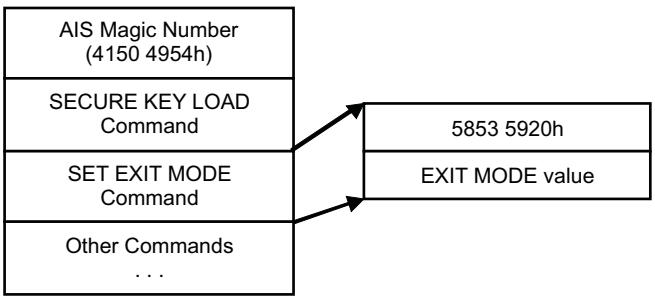


Table 12. Supported Exit Types

Device Type	NONSECURE	SECURE_WITH_SK	SECURE_NO_SK
Basic Secure Boot	Allowed	Allowed	Prohibited

Table 13. Values for Secure Boot Exit Types

Exit Type	Value
BOOT_EXIT_NONSECURE	0
BOOT_EXIT_SECURE_WITH_SK	1h
BOOT_EXIT_SECURE_NO_SK	2h

3.2.3 Signature Requirements

The integrity of the boot images on a basic secure boot device is assured using secure signatures. These signatures consist of encrypted secure hash values. A single boot image may contain multiple signatures, the reasons for which are detailed below. The hash values are generated over the contents of the boot image, starting from the first word following the end of the previous signature. The first signature to appear in a secure AIS boot image will contain a hash of the boot image contents starting from and including the AIS magic word.

Because the AIS image format can allow certain parts of the chip state to be altered during the boot process, care must be taken to make sure that no malicious attacker can insert new commands, or change the parameters of existing commands. For this reason, a single signature at the end of the boot image is not sufficient to protect the boot process.

Before any AIS command that can alter system state is executed, a signature check must take place. Furthermore, before any AIS commands that can possibly read system state can be allowed to execute, a signature check must take place. Finally, a signature check must always take place at the very end of the boot image.

3.2.3.1 Signatures for Basic Secure Boot Devices

Consider the example of the SET command on a basic secure boot device, shown in the following AIS fragment:

```
/* 0x00000000 */ 0x41504954, // AIS MAGIC
/* 0x00000004 */ 0x58535920, // Secure Key Load
/* 0x00000008 */ ----------  //   Key Data
/* 0x00000028 */ 0x58535907, // Set
/* 0x0000002C */ 0x00000002, //   Type
/* 0x00000030 */ 0x41000100, //   Address
/* 0x00000034 */ 0x00000000, //   Data
/* 0x00000038 */ 0x00000004, //   Delay
/* 0x0000003C */ ----------  //   Signature (for 0x00000000 - 0x0000003C)
```

Signatures for basic secure boot devices are encrypted using the AES-128 standard using the customer encryption key (CEK) that is loaded with the SECURE KEY LOAD command at the start of the boot process.

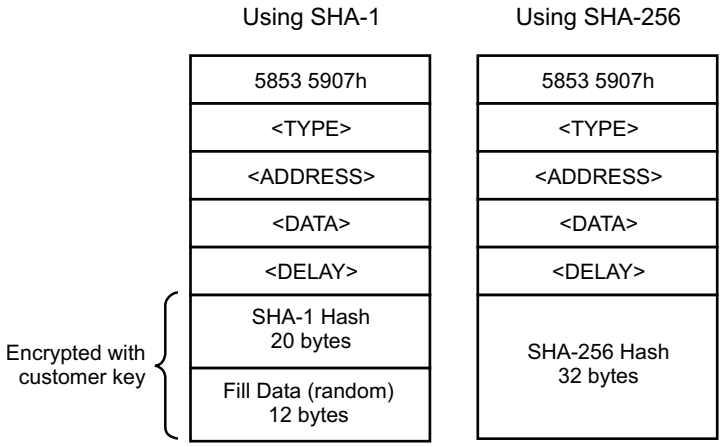
The following AIS fragment shows the usage of the JUMP and JUMP AND CLOSE commands:

```
/* 0x00000050 */ 0x58535904, // Disable CRC
/* 0x00000054 */ 0x58535922, // Encrypted Section Load
/* 0x00000058 */ 0x10000000, //   Address
/* 0x0000005C */ 0x000004A0, //   Size
/* 0x00000060 */             //   Data
/* 0x00000500 */ 0x58535905, // Jump
/* 0x00000504 */ 0x1000005C, //   Address
/* 0x00000508 */ ----------  //   Signature (for 0x00000050 - 0x00000508)
/* 0x00000528 */ 0x58535922, // Encrypted Section Load
/* 0x0000052C */ 0x1180F000, //   Address
/* 0x00000530 */ 0x00000600, //   Size
/* 0x00000530 */             //   Data
/* 0x00000B30 */ 0x58535922, // Encrypted Section Load
/* 0x00000B34 */ 0x1180F658, //   Address
/* 0x00000B38 */ 0x00000014, //   Size
/* 0x00000B3C */             //   Data
/* 0x00000B50 */ 0x58535922, // Encrypted Section Load
/* 0x00000B54 */ 0x1180F600, //   Address
/* 0x00000B58 */ 0x00000054, //   Size
/* 0x00000B5C */             //   Data
/* 0x00000BB0 */ 0x58535906, // Jump N Close
/* 0x00000BB4 */ 0x1180F480, //   Address
/* 0x00000BB8 */ ----------  //   Signature (for 0x00000528 - 0x00000BB8)
```

Again note that the signature data is only 32 bytes for the basic secure boot device.

The signature data consists of two AES-128 encryption blocks, for a total of 32 bytes (see Figure 16). When the SHA-1 hash algorithm is used, only 20 bytes of the 32 bytes contains hash data. The last 12 bytes of the 32 bytes should be filled with random values before encryption and are ignored by the boot loader upon decryption. When the SHA-256 hash algorithm is used, all 32 bytes are filled with the 256-bit hash value.

Figure 16. Example of SET Command with Signatures on Basic Secure Boot Device

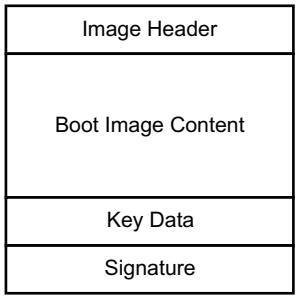


3.3 Non-AIS Boot Modes

The NOR legacy and UHPI boot modes do not use the AIS boot image format. Rather these allow the loading of a variable-size monolithic secondary boot loader image to the start of the L2 RAM. These modes are similar to their non-secure versions, but with important differences to support decryption and signature verification.

Both modes require the same format. The size of the image varies between 1KB and 16KB in 1KB increments. This size includes all required headers, keys, and signatures, so the actual size of the executable code is less than the total image size. Figure 17 shows the format of the boot image for these modes.

Figure 17. Image Format for Non-AIS Boot Images



The image header format is shown in Table 14. The copy word of the header is used in the same way as it is in the non-secure NOR legacy boot mode. Bits 11-8 contain a value indicating the number of kilobytes in the boot image. An entire image should always end on a 1KB boundary.

The magic word of the header indicates whether the boot image content is simply signed (5194 C0DEh) or if it is both signed and encrypted (034C C0DEh).

The boot entry point address is the address to which the boot loader will jump when the boot is complete. The secure data size is the combined size of the key data and the signature at the end of the image.

The key data in this image takes the exact same form as the key data used in the SECURE KEY LOAD AIS command (see Section 3.2.2.1). The signature takes the same form as those used in the AIS images (see Section 3.2.3). The ROM exit mode (see Section 3.2.2.3) for these non-AIS boot modes is NONSECURE on basic secure boot devices.

Table 14. Header Format for Secure Non-AIS Boot Images

Offset	Value
0	Copy word
4h	Magic Word (5194 C0DEh or 034C C0DEh)
8h	Boot entry point address
Ch	Secure Data Size

3.3.1 NOR Legacy Boot Mode

In the NOR legacy boot mode, the Copy Word of the image header is read from the start of the EMIFA CS2 region. The size of the boot image is read from that word. The entire boot image (including the copy word and the other header elements) is copied from the start of the EMIFA region to the start of the L2 RAM.

The secure data size value from the image header is then used to find the key data relative to the end of the image. The keys are installed first so that the secure hash algorithm is selected. If the magic word of the header indicates that the image is encrypted, the contents are then decrypted. The image header, image data, and key data are then hashed. Finally, the signature is decrypted and the hashes are compared to verify the signature. If everything checks out, the ROM boot loader exits to the entry point specified in the image header.

3.3.2 UHPI Boot Mode

The same image format is used for UHPI mode, but the manner in which it is delivered is different. The external host places the entire secure boot image into the L2 RAM directly, while the DSP waits for the DSPINT bit of the HPIC register to be set. This is the same mechanism used in the non-secure case.

After the image has been loaded by the external host, the DSP follows the same procedure used in the NOR legacy boot mode. The image size must still be a multiple of 1KB, up to 16KB, despite the fact that the external host can load more data than this.

3.4 Generating Secure AIS Boot Images

The tool used to generate secure AIS boot images is called SecureHexAIS. The tools must be specific to the family of devices being targeted for boot (OMAP-L138/C6748/C676/C6742 or OMAP-L137/C6747/C6745/C6743). The tool can process executable images of either ELF or COFF format or binary images and create a single AIS image in binary, C array, or S-record format.

Documentation for this tool should be provided in the software package containing the boot utilities.

3.5 Secondary Boot Loader

Note that the ROM boot loader can only process a single AIS image during the boot process. Once the ROM boot loader exits, the AIS parsing routines of the ROM are no longer available. Furthermore, the device exit type cannot be altered once the ROM boot loader has exited.

If a secure secondary boot loader is desired, then it is recommended that the SK_EXIT mode be used and the SK_decryptMod API called to decrypt/authenticate binary data in boot load module format.

4 Secure Kernel

The secure kernel is at the heart of the C674x DSP security system. This section provides an overview of the Secure Kernel operation.

4.1 Introduction

The Secure Kernel uses a combination of privilege levels, hardware features, and ROM-based software to provide security to both code and data.

- Hardware Management. Many C674x DSP core hardware features present security challenges. Cache modes, EDC hardware and so on all present interesting security complications. The C674x DSP core addresses this by restricting problematic functionality to Secure Supervisor access only, which implies that Secure Kernel must handle these functions.
- Software Interface. The Secure Kernel provides a mechanism for invoking its own APIs, and to allow a hosted operating system to load and execute applications, as well as secure algorithms.

4.2 Hardware Management

The Secure Kernel provides the following functionality

- Hardware Initialization. In conjunction with the secure ROM boot loader, the DSP core hardware is put into a known state after a reset.
- Interrupt and Exception Servicing. The only way to enter Secure Supervisor mode on a C674x DSP core is through a reset, interrupt, or exception. All these entry points are controlled by the Secure Supervisor ROM. Note that software exceptions are used to communicate between the non-secure application and the Secure Kernel and secure algorithms.
- Proxy Access to Hardware Features

4.3 Software Interface

All Secure Kernel API calls are invoked from non-secure code via the software exception (SWE) instruction. The Secure Kernel also provides a means to invoke host operating system calls via the downcall API. A more detailed discussion of this API can be found at SK_switchNonSec.

4.4 Stack Contexts

A stack context is a secure memory structure that is used to contain stack data during secure algorithm execution. The Secure Kernel maintains a pool of stack contexts to coordinate the switching between secure and non-secure stack with the host operating system. The host OS registers a single Context Word Pointer with the Secure Kernel. This pointer points to a single 32-bit value in non-secure data space that contains the currently active context number. The host OS is responsible for updating this context number whenever it switches the currently active task (and associated stack frame).

The Secure Kernel uses the registered Context Word Pointer to select the appropriate stack context when it needs to switch to a secure stack (typically when invoking a secure algorithm).

When allocating a secure context, the application registers the upper and lower bounds of the corresponding non-secure stack. The Secure Kernel will verify that the current stack pointer falls within the specified bounds, before switching to the secure stack. You can defeat this stack checking mechanism by specifying the entire memory range as legal stack bounds (0000 0000h-FFFF FFFFh).

5 Secure Kernel Application Programming Interface

This section provides a detailed description of the individual Secure Kernel API functions.

5.1 Secure Kernel API Calls

All Secure Kernel API calls, Table 15, are invoked via the Software Exception (SWE) instruction. Secure Kernel uses the following calling conventions:

- The system call number is passed in register B0. All Secure Kernel system call numbers are negative; non-negative numbers are reserved for the hosted operating system.
- Arguments in A4, B4, etc. per standard C calling convention.
- Return value (if any) in A5:A4, per standard C calling convention.
- Error status returned in A0. Zero (0) indicates no error.

NOTE: Exceptions to the standard C calling convention

The standard C calling convention calls for only the first 10 arguments to be passed through registers. Likewise, in variable argument function calls, the anchor (last explicitly declared argument) and following arguments are passed on the stack. Since Secure Kernel API functions switch to a secure stack during execution, these functions (including secure application functions) are limited to a maximum of 10 arguments, and do not support the variable argument calling conventions.

Secure functions that return a structure are also not supported.

System calls return error status separately from the function return value, to allow unambiguous interpretation of whether the call succeeded.

There is one important exception to the scheme above. The “downcall” system call—an API that allows Secure User to invoke a Non-secure Supervisor API—expects the hosted Operating System’s system call number to be in A0. This value will eventually get copied to B0. A0 is used so that all the other arguments remain in place in the register file.

A Secure Kernel library is provided that contains C-callable wrapper functions for all the externally visible APIs. These C interfaces are linked with the application and contain embedded knowledge of the assembly interfaces to the Secure Kernel.

Table 15. API Function List

API Function	Call Number	Function
SK_registerSCWP	-5	Stack Context Word Pointer (SCWP) Registration
SK_allocSC	-13	Stack Context (SC) Allocation
SK_freeSC	-14	Stack Context (SC) Freeing
SK_switchNonSec	-22	Switch to non-secure operating mode
SK_setUserKey	-23	Encrypt user key with KEK (basic secure boot)
SK_setJTAGControl	-24	Control Debug TAPs
SK_decryptMod	-26	Authenticate and optionally decrypt boot load module

5.2 API Functions

This section contains the API reference pages for the C wrappers. All wrapper functions are either a void return for APIs that cannot fail, or return a signed integer that overloads valid return values with error codes in the negative domain.

SK_registerSCWP *Stack Context Word Pointer (SCWP) Registration*

Syntax

```
int SK_registerSCWP(scwp, numSC);
```

Parameters

`int *scwp` – pointer to an integer owned by the Operating System that tracks the current secure context.

`unsigned numSC` – number of secure contexts into which the single secure stack block is divided.

Return Value

`SCWP`, if no error

`SK_EPRIV` if not called from supervisor privilege level

`SK_EINUSE` if any secure contexts are currently allocated

`SK_EINVAL` if `scwp` is not word-aligned

`SK_ECTX` if `numSC` is bad (can only be 1, 2, 4, or 8)

Constraints

Must be called from supervisor privilege level, and scwp must be word-aligned. All stack contexts must be freed before calling this API.

`SK_registerSCWP()` is used by an Operating System to register the address of an integer that contains the current stack context identifier. It also takes a parameter containing the desired number of stack contexts for the Secure Kernel to manage, in order to control the stack size of each stack context.

The `scwp` parameter points to an integer that will be updated by the Operating System during context switches. Each thread that needs to make a stack-based system call (`SK_load()`, `SK_algoInvoke()`) must have an associated stack context reserved for it (through `SK_allocSC()`), and the stack context word (`SCW`) must be written with the stack context identifier for the current thread. A value of -1 is used for threads that do not have an associated stack context.

The `numSC` parameter informs the Secure Kernel how many stack contexts to manage. This is used to control the size of the stack for each secure context. The Secure Kernel manages a single block of memory for all secure stacks, and this block can be subdivided into 1, 2, 4 or 8 secure contexts. With one context, the whole stack block is used for the single secure context. With eight contexts, the stack block is divided into eight equal-sized blocks, one for each stack context. This support allows for a large stack when needed, while allowing for more secure contexts when needed.

SK_allocSC *Stack Context (SC) Allocation*

Syntax

```
int SK_allocSC(nsStackBottom, nsStackTop);
```

Parameters

`unsigned nsStackBottom` – lower address of non-secure stack

`unsigned nsStackTop` – upper address of non-secure stack (+ 1)

Return Value

secure context ID, if no error

`SK_EID` if no secure contexts are currently available

Constraints

The `SK_allocSC()` API is intended for use by an Operating System to obtain a stack context for use while non-secure user code calls secure code (either just a Secure Kernel API, or a secure algorithm). A “stack context” is an internal data structure, not visible outside of secure space, used by the Secure Kernel to record secure stack information for the purpose of multithreading secure threads. It contains a secure stack that is used while secure code is running.

The two parameters to `SK_allocSC()` define a non-secure stack block for the allocated context. This indicates to the Secure Kernel the range of the non-secure stack pointer to expect when the allocated context is in use. The Secure Kernel checks that the non-secure stack pointer is within this range when returning from a downcall. The stack parameters for a specific TSK object can be obtained using the task->stack and task->stacksize (in bytes) values.

The return value is an integer index into the internal Secure Kernel secure context table. The Operating System must write this integer into a dedicated word (see SK_registerSCWP API) to indicate the current stack context to be used when secure code is invoked. Each context should be logically attached to a specific thread, although multiple non-secure threads may share a single context (with stack checking turned off).

SK_freeSC *Stack Context (SC) Freeing*

Syntax

```
int SK_freeSC(contextID);
```

Parameters

`int contextID` – secure context ID obtained with a call to `SK_allocSC`

Return Value

`SK_EID` if passed `contextID` is not valid

Description

The `SK_freeSC()` API is used by an Operating System to free a stack context that was previously allocated with `SK_allocSC()`.

SK_switchNonSec *Switch to non-secure mode*

Syntax

```
void SK_switchNonSec(entryAddr);
```

Parameters

`void *entryAddr` – pointer to entry address of code to execute after secure to non-secure override.

Return Value

none

See Also

`SK_setJTAGControl()`

Description

The `SK_switchNonSec` API is used to switch the DSP operating mode to non-secure. All secure information (including the secure boot ROM) will disappear, and the program counter will start at the indicated address after the mode switch. The mode switch involves a soft reset of the DSP power domain and an opening of all chip-level security features.

This routine will also freeze the JTAG TAPs in the disabled state, unless they have already been enabled/disabled before.

The override sequence uses the first 32 bytes of the DSP L2 memory region to store information that must survive the DSP reset. The user application should not use this memory if use of the `SK_switchNonSec` API is required.

SK_setUserKey *Encrypt user key structure with random key*

Syntax

```
void SK_setUserKey(keyStruc);
```

Parameters

`void *keyStruc` – pointer to the user key structure. This structure must be located in non-secure writeable memory.

Return Value

`SL_OK` if successful

`SL_BADDEVTYPE` if this API has been called once already

`SL_EINVAL` if any other error occurred

Description

The `SK_setUserKey` API is used to encrypt the user key structure for basic secure boot devices.

The user key structure consists of two AES packets (32 bytes):

```
struct SL_usrKeyStruc {
    int usrKeyMagic;       /* user key structure magic number     */
    int JTAGforceOff;      /* flag to force TAPs off permanently  */
    int hashAlgSelect;     /* selected SHA algorithm              */
    int randomSeed[1];     /* random values                       */
    int custKey[4];        /* actual customer key value           */
}
```

`usrKeyMagic` must be initialized with the value BE40 C0DEh.

The `JTAGforceOff` field, when set to non-zero, is used to force the JTAG TAPs to remain turned off. If this field is zero, the JTAG TAPs can still be controlled using the `SK_ setJTAGControl()` API call.

The `hashAlgSelect` field contains the selected SHA algorithm value: 0 = SHA-1, 1h = SHA-256, 2h = SHA-384, 3h = SHA-512. Currently, only SHA-1 and SHA-256 are supported on this device, so this field should be set to 1 or 0.

The `randomSeed[ ]` field can be set to any value. It is used to make breaking the encrypted user key more difficult.

The `custKey[ ]` field contains the 128-bit customer encryption key.

Upon successful return, the now KEK encrypted structure contains the specified customer key value in a form that can be used by the ROM boot loader to set the internal customer key securely.

The user key structure is encrypted using AES in CBC mode, with a ROM defined initialization vector. This API can only be called once per boot to help prevent force attacks on the KEK.

SK_setJTAGControl *Enable / disable JTAG – Control Debug TAPs*

Syntax

```
void SK_setJTAGControl(controlWord);
```

Parameters

`unsigned int controlWord` – Word to control the bits of the debug TAPs control register.

Return Value

none

Description

The `SK_setJTAGControl` API is used to permanently enable or disable the Debug TAPs (including JTAG TAP). Note that, since this operation is performed through a write-once register, once the JTAG TAPs have been disabled, only a power-on reset (POR) can restore the register to a writeable state. Also note that this a run-time API, and it is possible that the TAPs were set up at an earlier stage of the system start-up process (for example, during boot).

SK_decryptMod *Authenticate and decrypt (if needed) a boot load module*

Syntax

```
int SK_decryptMod(pBuffer, size);
```

Parameters

`void *pBuffer` – pointer to memory location containing boot load module data

`int size` - load module size (includes payload size + header size + signature size)

Return Value

`SL_DECRYPTFAIL` if payload size is too small

`SL_BADMODHDR` if the header is not valid (due to decryption failure or otherwise)

`SL_BADCOMMAND` if the command is not allowed

`SL_CHECKSIGFAIL` if the signature validation fails

Description

The `SK_decryptMod` API processes a boot load module that is already in memory. The API examines the load module magic number (in the boot load module header) to determine if the boot load module is encrypted or not. If the magic number is in the clear (reads 70AD C0DEh), then no decryption is performed. If it is not in the clear, then the boot load module header is decrypted in AES CBC mode (the IV is the encryption key encrypted by the encryption key). Decryption is attempted with the current installed encryption key (either the CEK or a delegate encryption key). If the magic number does not appear, then decryption is attempted using the KEK. If the magic number still does not appear, then the function returns `SL_BADMODHDR`.

If the decryption of the load module header succeeds, then the decrypted load module header is written back to memory (during the tests for the magic number, all decryption happens in local secure memory). The AES CBC decryption then continues with the payload data at the start of the buffer. Note that the CBC chain starts with the load module header and then continues with the payload data. The decryption happens as if the encrypted data buffer is organized as a load module header followed by payload – but the physical layout of the load module buffer is payload followed by the load module header. This is important to keep in mind when performing the encryption offline. It is also important to note that the boot load module payload size must be a multiple of the AES block size.

After all payload and header data is available as plaintext (either decryption succeeded or no decryption was required), the signature is validated. The decrypted load module contents are hashed using the secure hash algorithm selected at boot time. The hashing operation is performed on the physical memory layout – first payload data and then load module header. Then the signature is decrypted and the calculated hash is compared against the stored hash to validate the loda module. If the check fails, the function returns `SL_CHECKSIGFAIL`.

The `SK_decryptMod` API is no longer callable after the `SK_switchNonSec` API has been called.

Table 16. Boot Load Module Structure for Basic Secure Boot Devices

	Size		
Name	**Bits**	**Bytes**	**Description**
PAYLOAD	Variable	Variable	Monolithic payload data
HEADER	128	16	Boot Load Module Header Structure
SIGNATURE	256	32	Signature data

Table 17. Header Structure for Boot Load Module

Name	Size		Description
	Bits	Bytes	
magicNum	32	4	Magic number for boot load module (70AD C0DEh)
size	32	4	Size of boot load module (payload + header + signature)
randomSeed	64	8	Random Values (unused)

5.3 Error Codes

Table 18 and Table 19 describe all general error codes returned from the Secure Kernel external APIs. Error codes from Secure Kernel system calls are returned in register A0.

Table 18. Error Codes for Secure Kernel APIs

Mnemonic	Value	Description
SK_ESTACK	-1	Bad secure / non-secure stack
SK_ECTX	-2	Bad secure context ID
SK_ESECLEVEL	-3	API called from wrong security level
SK_ELOADMAGIC	-4	Bad load module header magic number
SK_ESECTMAGIC	-5	Bad load module section header magic number
SK_EDECODE	-6	Load module decoding error
SK_EINUSE	-7	Load module memory already in use
SK_EID	-8	No available or bad load module ID
SK_EEP	-9	No available load module entry point slot
SK_EEPINVALID	-10	Bad entry point index for SK_algoInvoke API
SK_EPRIV	-11	Bad privilege level
SK_EBADRPK	-12	Bad Root Public Key
SK_EBADSIGN	-13	Bad signature
SK_EINVAL	-14	General invalid value

Table 19. Error Codes for Secure Kernel Load APIs

Mnemonic	Value	Description
SL_OK	0	No error
SL_BADRPK	1	Bad Root Public Key
SL_BADMODHDR	2	Bad boot Load module header
SL_BADKEYSIGN	3	Not used
SL_DECRYPTFAIL	4	Bad payload size (too small) for load module
SL_BADMEMCONFIG	5	Not used
SL_BADCOMMAND	6	Command prohibited
SL_EINVAL	7	Invalid input value
SL_BADDEVTYPE	8	Basic Secure Boot Key encrypt not allowed
SL_CHECKSIGFAIL	9	Bad signature

6 Configuration Files

This section describes the configuration file for the Secure Kernel based on the tie-offs for the C674x core, and the C wrapper files required to use the Secure Kernel API within non-secure applications.

6.1 C Wrapper Library File

The C wrapper library file, `sk_cwrap.asm`, contains the source code for the wrapper functions required to call the Secure Kernel API. Note that `SK_eventMap()` is not a pure API function; it actually maps into the `SK_setEventReg()` API call.

Example 1. C Wrapper Library File

```
sect    ".text"

SK_SYSCALL_REGSCWP        .set -5
SK_SYSCALL_ALLOCSC        .set -13
SK_SYSCALL_SWITCHNONSEC   .set -22
SK_SYSCALL_SETUSERKEY     .set -23
SK_SYSCALL_SETJTAGENABLE  .set -24
SK_SYSCALL_GETSECKEY      .set -25
SK_SYSCALL_DECRYPTMOD     .set -26


syscall .macro    funcName, callNum
        .global funcName
funcName:
        MVK    callNum, B0
||      BNOP   syscallCommon, 5
        .endm

;;
;; Common code for syscall wrappers.  Should be called with the syscall
;; code in B0.
;;

syscallCommon:
        STW    B3, *B15--[2]
        SWE
        NOP

        LDW    *++B15[2], B3
[ A0]   NEG    A0, A4            ; A0 - 0 success, non-zero failure
        NOP    3
        BNOP   B3, 5

;;
;; _SK_downcallOS is defined in sk_swe.asm, since it needs to exist
;; in secure space to be callable by a secure algo.
;;

      syscall _SK_registerSCWP,         SK_SYSCALL_REGSCWP
      syscall _SK_allocSC,              SK_SYSCALL_ALLOCSC
      syscall _SK_switchNonSec,         SK_SYSCALL_SWITCHNONSEC
      syscall _SK_setUserKey,           SK_SYSCALL_SETUSERKEY
      syscall _SK_setJTAGControl,       SK_SYSCALL_SETJTAGENABLE
      syscall _SK_getSecKey,            SK_SYSCALL_GETSECKEY
      syscall _SK_decryptMod,           SK_SYSCALL_DECRYPTMOD

              .end
```

6.2 C Wrapper Header File

The header file, `cwrap.h`, should be included in all programs making Secure Kernel API calls. It provides the prototypes for all Secure Kernel API functions.

Example 2. C Wrapper Header File

```
/* ================================================================
 *
 *  (c) Copyright Texas Instruments, Incorporated. All Rights Reserved
 *
 * Use of this software is controlled by the terms and conditions found
 * in the license agreement under which this software has been supplied.
 * ================================================================ */
/** @file sk_cwrap.h
 *
 * @version 01.00
 */
/* ================================================================
 * Revision History
 * ================
 *
 * ================================================================ */

#ifndef __SK_CWRAP_H
#define __SK_CWRAP_H

#ifdef __cplusplus
extern "C" {
#endif

typedef unsigned SK_LoadObj;

int  SK_registerSCWP(int *scwp, unsigned numSC);
int  SK_allocSC(unsigned nsStackBottom, unsigned nsStackTop);
int  SK_freeSC(int contextID);
void SK_switchNonSec(void *entryAddr);
int  SK_setUserKey(void *keyStruct);
void SK_setJTAGControl(unsigned int controlWord);
int  SK_decryptMod(void *pBuffer, int size);

/*
 * General error codes returned from SK C interfaces.
 */
#define SK_ESTACK       -1      /* bad secure/non-secure stack      */
#define SK_ECTX         -2      /* bad secure context ID            */
#define SK_ESECLEVEL    -3      /* API called from wrong sec        */
#define SK_ELOADMAGIC   -4      /* bad algo load header magic       */
#define SK_ESECTMAGIC   -5      /* bad algo sect header magic       */
#define SK_EDECODE      -6      /* algo decoding error              */
#define SK_EINUSE       -7      /* algo memory already in use       */
#define SK_EID          -8      /* no available or bad algo ID      */
#define SK_EEP          -9      /* no available algo EP slot        */
#define SK_EEPINVALID   -10     /* bad entry point index for invoke */
#define SK_EPRIV        -11     /* bad privilege level              */
#define SK_EBADRPK      -12     /* bad Root Public Key              */
#define SK_EBADSIGN     -13     /* bad signature                    */
#define SK_EINVAL       -14     /* general "invalid value"          */

#ifdef __cplusplus
}
#endif

#endif /* __SK_CWRAP_H */
```

7 Encrypting a Basic Secure Boot Device Key

For basic secure boot devices, the device decryption key needs to be encrypted by the device it needs to run on. This section describes how to do this using a number of Secure Kernel commands.

7.1 Code Prolog

On basic secure boot devices, you need to provide a decryption key to be able to decrypt signatures and encrypted data. This key itself is normally encrypted with the device specific KEK. To encrypt your key initially, you need to boot into the Secure Kernel, and use a number of Secure Kernel API calls. This section describes the software necessary to do this.

The code prolog shown in Example 3contains the typical information necessary for an application that uses the Secure Kernel API. All API function prototypes are defined in the file `sk_cwrap.h`, and a definition for the user key structure is provided.

Finally, we define the stack context word (scw) that must be provided to the Secure Kernel for secure stack context management. The values `_stack` and `_STACK_SIZE` will be provided by the linker to allow for stack checking when switching stack contexts.

Example 3. Code Prolog

```
/*! ===========================================================================
 *!
 *!  (c) Copyright Texas Instruments, Incorporated. All Rights Reserved
 *!
 *! Use of this software is controlled by the terms and conditions found
 *! in the license agreement under which this software has been supplied.
 *! ========================================================================== */
/** @file keyFragment.c
 *
 * @version 01.00
 */
/*! ===========================================================================
 *! Revision History
 *! ================
 *!
 *! ========================================================================== */

/* ============================================================================
 * INCLUDE FILES
 * ========================================================================== */
/* --------------------- system and platform files -------------------- */
#include <stdio.h>
#include <stdint.h>
/* -------------------------- program files ---------------------------- */
#include "sk_cwrap.h"

#define GENKEY_MAGIC    0xBE40C0DE

typedef struct SL_usrKeyStruc {
    int usrKeyMagic;
    int JTAGforceOff;
    int hashAlgSelect;
    int randomSeed;
    long long custKey[2];
} SL_usrKeyStruc;

/* ============================================================================
 * EXTERNAL REFERENCES
 * ========================================================================== */
/* ------------------------ data declarations ------------------------- */
#pragma DATA_SECTION(scw, ".far:_main")
far int scw;

extern far uint32_t _stack;
extern far uint32_t _STACK_SIZE;
```

7.2 Application Fragment

In your application you create (or load from flash) an unencrypted user key structure, similar to the one shown in the code section in Example 4.

Example 4. Initialization Code Fragment

```
void main(void)
{
    uint32_t stackBottom, stackEnd;
    SL_usrKeyStruc userKey;

    userKey.usrKeyMagic   = GENKEY_MAGIC;
    userKey.JTAGforceOff  = 0;
    userKey.hashAlgSelect = 0;
    userKey.randomSeed    = 0xdeadbeef;
    userKey.custKey[0]    = 0x0123456789abcdefL;
    userKey.custKey[1]    = 0xfedbca9876543210L;

    if (setUserKey(&userKey) != 0) {
        // error handling - something went wrong
    } else {
        // user key has been encrypted - write back
    }

    return;
}
```

The resulting structure for this example looks as follows in memory:

```
0xBE40C0DE 0x00000000 0x00000000 0xDEADBEEF
0x89ABCDEF 0x01234567 0x76543210 0xFEDBCA98
```

Since we are dealing with a little-endian processor, the programmed key value for this sample structure (`custKey[0]` and `custKey[1]`) is:

```
EF CD AB 89 67 45 23 01 10 32 54 76 98 CA DB FE
```

The `JTAGforceOff` field is programmed to 0, leaving the JTAG TAP control open for modification. In a typical deployment, this value would usually be set to a non-zero value to disable the JTAG TAPs.

The `hashAlgSelect` field indicates that the SHA-1 algorithm is used for all hash computations (use the value 1 for SHA-256).

The `randomSeed` field can be set to any value you want; it is not used during key verification.

The code fragment for the `setUserKey()` function is shown in Example 5. This function must be executed under control of the Secure Kernel. It combines all calls to the Secure Kernel necessary to encrypt the user key structure.

The `setUserKey()` function uses the following Secure Kernel API calls:

- `SK_registerSCWP()` to register the Stack Context Word Pointer (see SK_registerSCWP)
- `SK_allocSC()` to allocate a stack context for the SK_setUserKey() call (see SK_allocSC)
- `SK_setUserKey()` to encrypt the user key structure (see SK_setUserKey)

If any API call fails, the value -1 is returned. A `setUserKey()` return value of 0 indicates that the user key structure has been successfully encrypted, and can be written back to the boot device.

Example 5. setUserKey() Function

```
int setUserKey(SL_usrKeyStruc *pUserKey)
{
    int resultVal;

    resultVal = SK_registerSCWP(&scw, 2);
    if (resultVal != &scw) {
        return -1;
    }

    stackBottom = (uint32_t) &_stack;
    stackEnd = stackBottom + (uint32_t) &_STACK_SIZE;

    scw = SK_allocSC(stackBottom, stackEnd);
    if (scw < 0) {
        return -1;
    }

    resultVal = SK_setUserKey((uint32_t) pUserKey);
    if (resultVal < 0) {
        return -1;
    }

    return 0;
}
```

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Audio	www.ti.com/audio	Communications and Telecom	www.ti.com/communications
Amplifiers	amplifier.ti.com	Computers and Peripherals	www.ti.com/computers
Data Converters	dataconverter.ti.com	Consumer Electronics	www.ti.com/consumer-apps
DLP® Products	www.dlp.com	Energy and Lighting	www.ti.com/energy
DSP	dsp.ti.com	Industrial	www.ti.com/industrial
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Security	www.ti.com/security
Logic	logic.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Power Mgmt	power.ti.com	Transportation and Automotive	www.ti.com/automotive
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com	Wireless	www.ti.com/wireless-apps
RF/IF and ZigBee® Solutions	www.ti.com/lprf		
	TI E2E Community Home Page		e2e.ti.com

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265