HASPOC
Secure Boot
Design Specification

Oniteo AB, Jan Stenbecks torg 17, 164 40 Kista, Sweden
Ph. +46 8 4441360, E-mail info@t2data.com www.t2data.com

19 September 2016

Revisions

Revision Overview

Revision	Log Message	Author	Date
PA1	Migrated to Document Framework	Hans Thorsen	2016-08-10
PA2	Update purpose description	Hans Thorsen	2016-08-12
PA3	Adjusted Purpose for modules	Hans Thorsen	2016-08-12

Table of Content

Table of Contents

Chapter 1 .

Modules

Secure Boot

Package	Description
bios_fip	Support for Firmware Image Package
bios_high	Secure Bios high level
bios_low	Arm Trusted Firmware 1.1 Adopted for HASPOC
bios_memory	SDRAM setup for bios
bios_output	Low level print package
bios_trust_anchor	interface for trust anchor
plf_bios_arm	Implementation of portability layer for ARM
secure_bios	Cryptograhic library targeted for firmware, boot parameter interface
spdlib	Artemis SPD Adapted for HSBF

Chapter 2 .

TOE Subsystem

The Secure Boot subsystem is divided into 9 modules. This document describes the purpose of each module, how they interact internally, as well as externally.

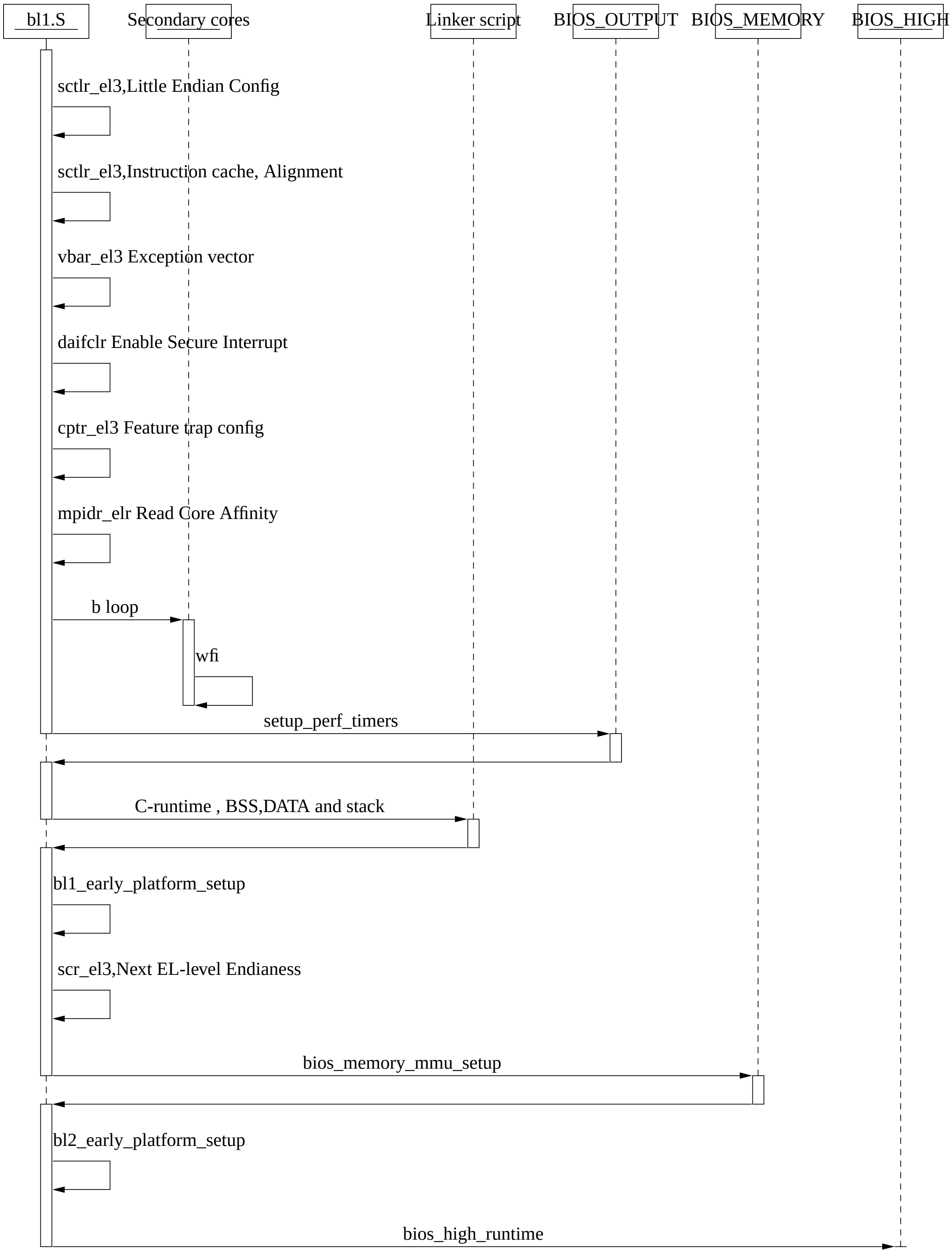
Portability

The modules are devided into two categories, platform specific and generic. Another benefit with portable modules is testing.

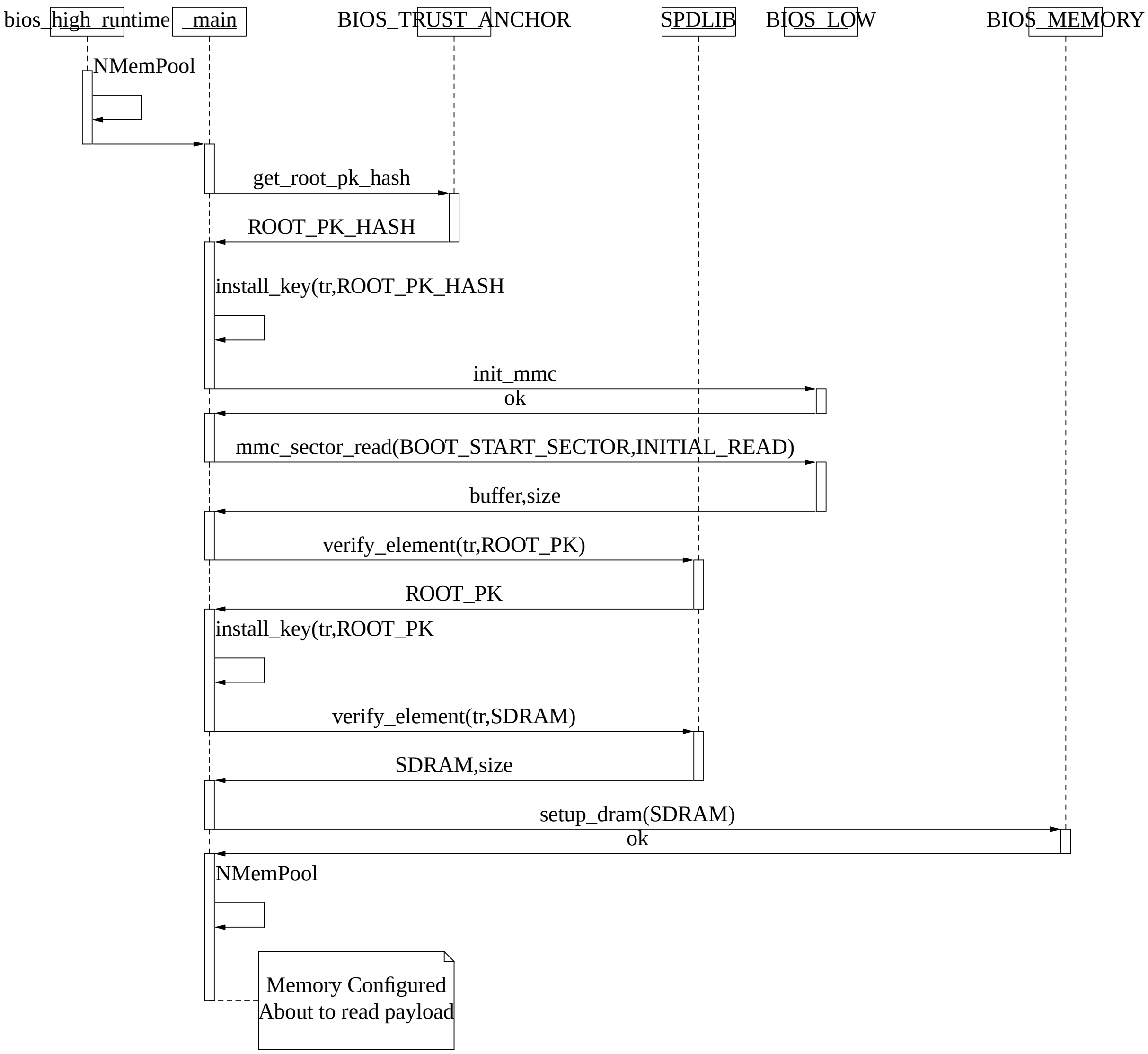
Platform specific modules (bios_low)	Portable generic modules (bios_high)
bios_low	bios_high
plf_bios_arm	spdlib
bios_memory	secure_bios
bios_trust_anchor	
bios_fip	
bios_output	

Interaction

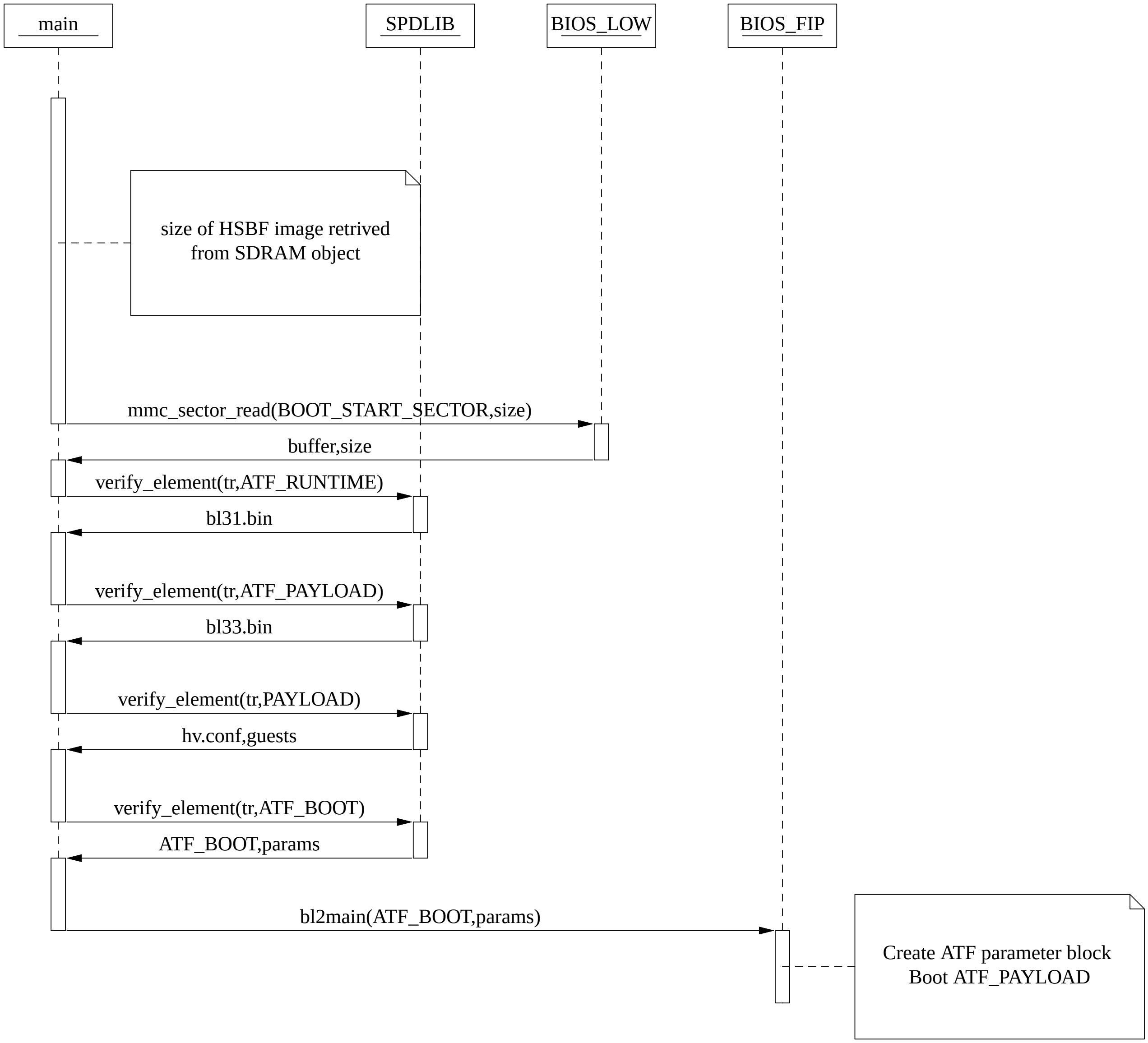
Execution in the Secure Boot starts with hardware reset and ends when control is passed to the hypervisor. The execution flow is illustrated with sequence diagram. The first diagram represents pure configuration that requires non portable assembler code (bios_low).
Next diagram retreives and verifies the signed HSBF image. Since the hardware executes in a very constrained environenment it only process the first objects required to establish a trusted boot and configuration of dynamic memory.
The following diagram read the entire HSBF image into dynamic memory and then process each object by verifying signature and then copy data to addresses found in the object header. The final phase is to create a parameter area in secure memory that contain addresses and execution parameters for the hypervisor.
The last diagram illustrates how Secure Boot invokes a forreign module (Arm Trusted Firmware runtime module / bl31.bin) that starts the hypervisor. The runtime modules install interrupt driven services that responds to power management of secondary cores.



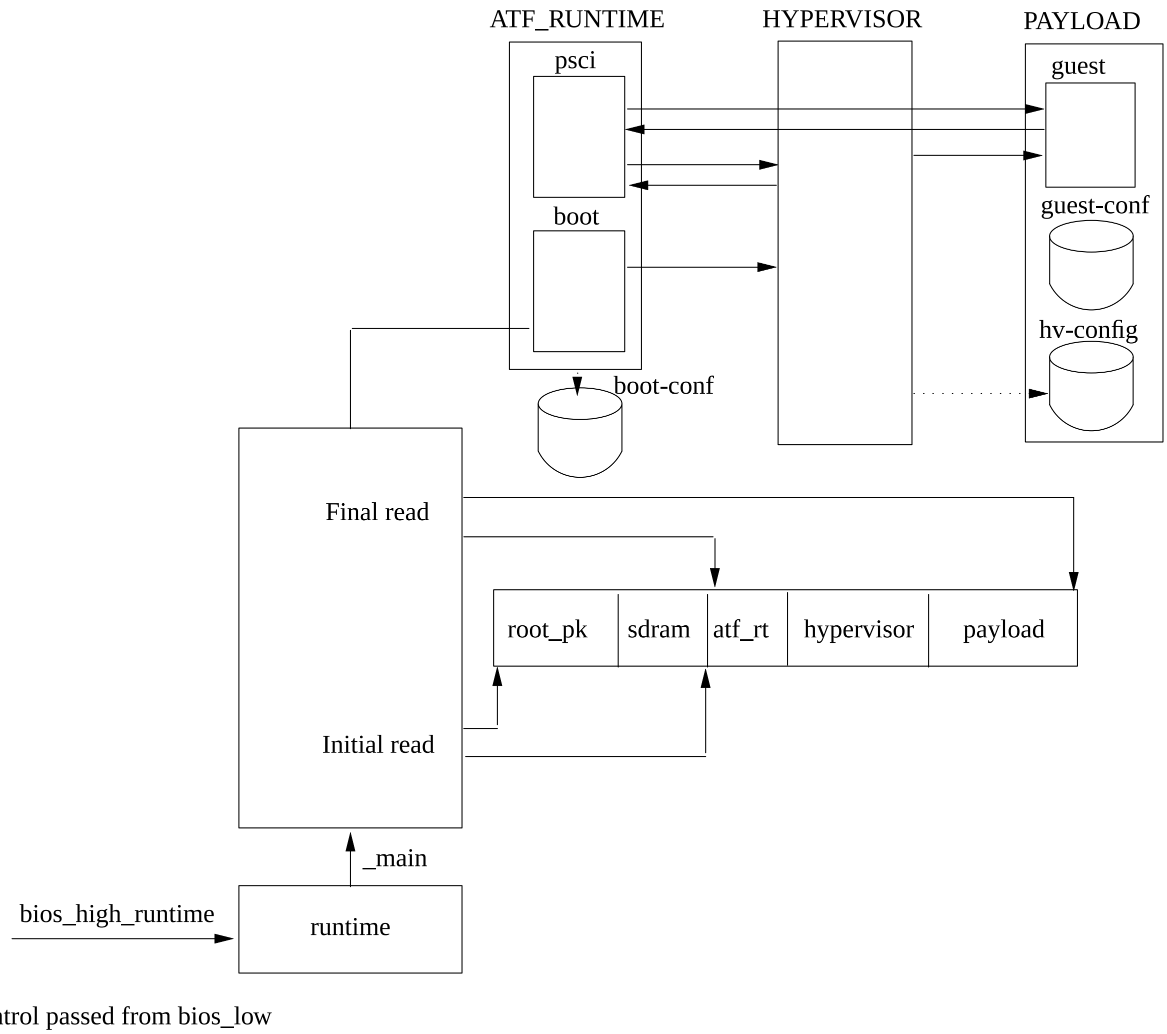
Sequence diagram for low level modules



Sequence diagram for high level (phase 1)



Sequence diagram for high level (phase 2)



Overview illustrating how bios_high invokes hypervisor

Chapter 3 .

Module bios_low

Purpose

This module configures hardware during boot and provides interface to storage.

Hikey boot

The Hikey development board is programmed in ROM to start execution in 32 bit monitor mode. Since Secure Boot is a true 64 bit application, there must be a small application written in 32 bits assembler that configures the ARMv8 processor to change state to 64 bit and exception level 3.
The source code could be downloaded from the URL below.
http://github.com/96boards/l-loader

```
Switch to aarch64 mode. CPU0 executes at 0xf9801000!
```

The first instruction in Secure Boot MUST reside in address 0xf9801000.
Since ARMv8 support many modes of operation, some registers must be written with to assure correct operation.
ARM supports both big and little endianness, therefore the correct value must be set prior to any access
against memory. The System Control Register for Exception level 3 (sctlr_el3) controls this propery.
The same register also controls instruction cache and alignment checks.

Attached peripherals could generate spurious interrupts, therefore a minimal interrupt handler must be defined and installed by writing the address of the vector to register vbar_el3.
To avoid concurrency related problems, all cores except one must be suspended. By reading the affinity register the primary core could be detected, all other core are calls Wait for Interrupt in a infinitive loop.
The runtime environment is established by

- Writing zeros to all address in the BSS region
- Copy global data from ROM to RAM.
- Assign stackpointer a address range.

The design of the configuration conforms to the model of Arm Trusted Firmware. Finally bios_low transfer control to module bios_high.

Support functions
This module provides interface for

- Disable and clean instruction cache
- Manage Power control
- Reading sectors from internal SD card (eMMC)

Interface

disable_mmu_icache_el3

Disable MMU and instruction cache

void disable_mmu_icache_el3(void)

init_mmc

This functions configures mmc drivers
needed for access of SD drivers

int init_mmc(void)

Return Codes

- MMC_INIT_OK
- MMC_INIT_ERROR

hisi_mcu_enable_sram

The actual purpose of this function is poorly
documented by Hikey.

The implementation writes
values to 32 bit registers that power on
memory needed to manage mcu.

void hisi_mcu_enable_sram(void)

sector_read

This functions retrieves sectors from a SD card.
Actual constraint defined by avaliable memory for target

int sector_read(sector, nr, buffer)

Name	Direction	Type	Purpose
sector	in	unsigned int	to start read from
nr	in	unsigned int	number of sectors to read
buffer	in	unsigned char *	to write retrieved data

Return Codes

- SECTOR_READ_OK
- SECTOR_READ_ERROR

hisi_mcu_start_run

The actual purpose of this function is poorly
documented by Hikey.

Some registers that controls ddr remap configuration

void hisi_mcu_start_run(void)

hisi_mcu_load_image

The bl30.image contains proprietary code that is
loaded into a dedicated region of memory

The actual copying is divided into sections.
prior to copying a section its headers is verified
if not valid it will return an error code

int hisi_mcu_load_image(_image_base, _image_size)

Name	Direction	Type	Purpose

Return Codes

- MCU_LOAD_OK
- MCU_LOAD_ERROR

Chapter **4** .

Module bios_output

Purpose

This module provides optional functions like output and timing.

Interface

asm_print_hex

Early print used in assembler code

void asm_print_hex()

Name	Direction	Type	Purpose
	in	n	hex digit

bios_output_setup_perf_timers

This function reset the performance counters
needed to timestamp boot performance

void bios_output_setup_perf_timers(void)

console_init

Setup serial port

void console_init(void)

printf

Write the output under the control of a format
string that specifies how subsequent arguments (or arguments accessed
via the variable-length argument
Write one character on serial port

int printf(format, ...)

Name	Direction	Type	Purpose
format	in	const char *	string specifying output

Return Number of characters printed upon success

asm_print_str

Early print used in assembler code

void asm_print_str()

Name	Direction	Type	Purpose
	in	s	string

Chapter 5 .

Module bios_memory

Purpose

Caching is required for performance when verifying signatures. ARMv8 requires MMU to be configured for caching to work.

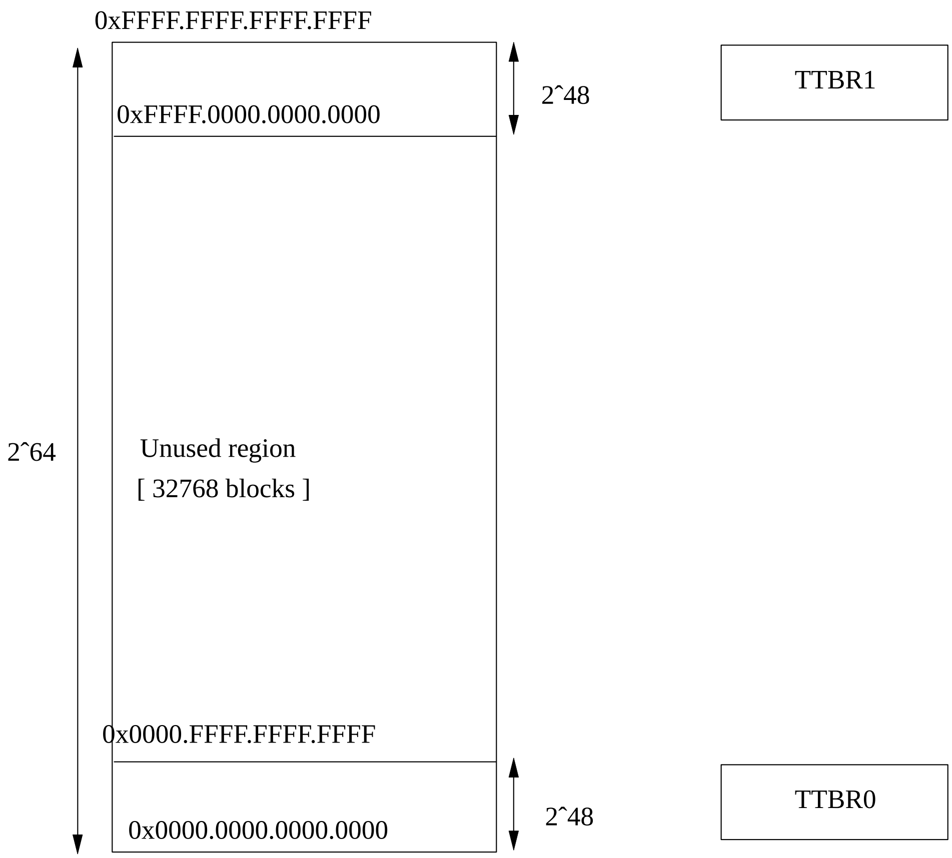
ARMv8 MMMU

This module update some system registers.

Register Name	Purpose
TTBR0_EL3	Translation Table Base Register
TCR_EL3	Translation Control Register
MAIR_EL3	Memory Attribute Indirection Register
SCTLR_EL3	System Control Register (enable/disable MMU)

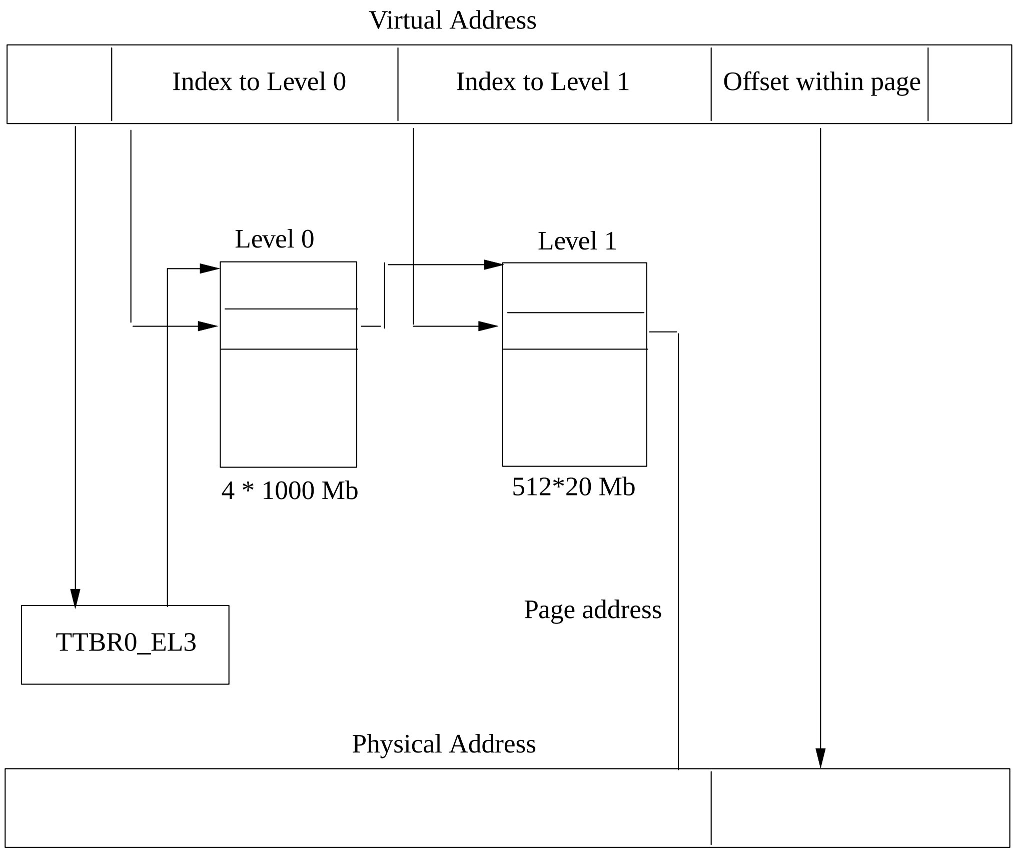
Address range

The actual implementation of the 64 bits architecture limits the available address range into two regions spanning 48 bits each. The usage of each region is defined in translation tables being associated with the Translation Table Base Register.



Address translation

Without MMU translation between physical and virtual address maps on to one. The boot process for the entire platform involves several steps (chain loading) and sometimes the MMU must be disabled, therefore the translation must be designed to preserve one-to-one translation when MMU is enabled. ARM supports translation in several stages.



The granularity of each translation table is defined in register TCR_EL3. To support legacy software written for 32 bit mode, the MMU should be restricted to the adress range of this mode. Chain loaders also allows the MMU to be reconfigured during the boot process and is therefore not a limiting factor for the entire system.

Memory attributes

Memory mapped IO appear as read/write memory, but caching must be disabled for devices. Each entry in the translation table contains an index into a array with cache properties. For Secure Boot two entries are required (memory and devices).

Interface

dcsw_op_level1

Cache operation level 1

void dcsw_op_level1(void)

dcsw_op_level2

Cache operation level 2

void dcsw_op_level2(void)

setup_sdram

This is a placeholder function that may
be used to configure memory.
The bl30.bin image retrived from storage
is passed as start and size.

int setup_sdram(start, size)

Name	Direction	Type	Purpose
start	in	uint64_t	of buffer
size	in	uint32_t	of buffer

flush_dcache_range

Assure that all data is written to
memory prior to boot of payload

void flush_dcache_range(start, size)

Name	Direction	Type	Purpose
start	in	uint64_t	of region to flush
size	in	uint64_t	of region

dcsw_op_all

High level cache operation

void dcsw_op_all(void)

bios_memory_mmu_setup

Secure Boot use one translation table
during all phases

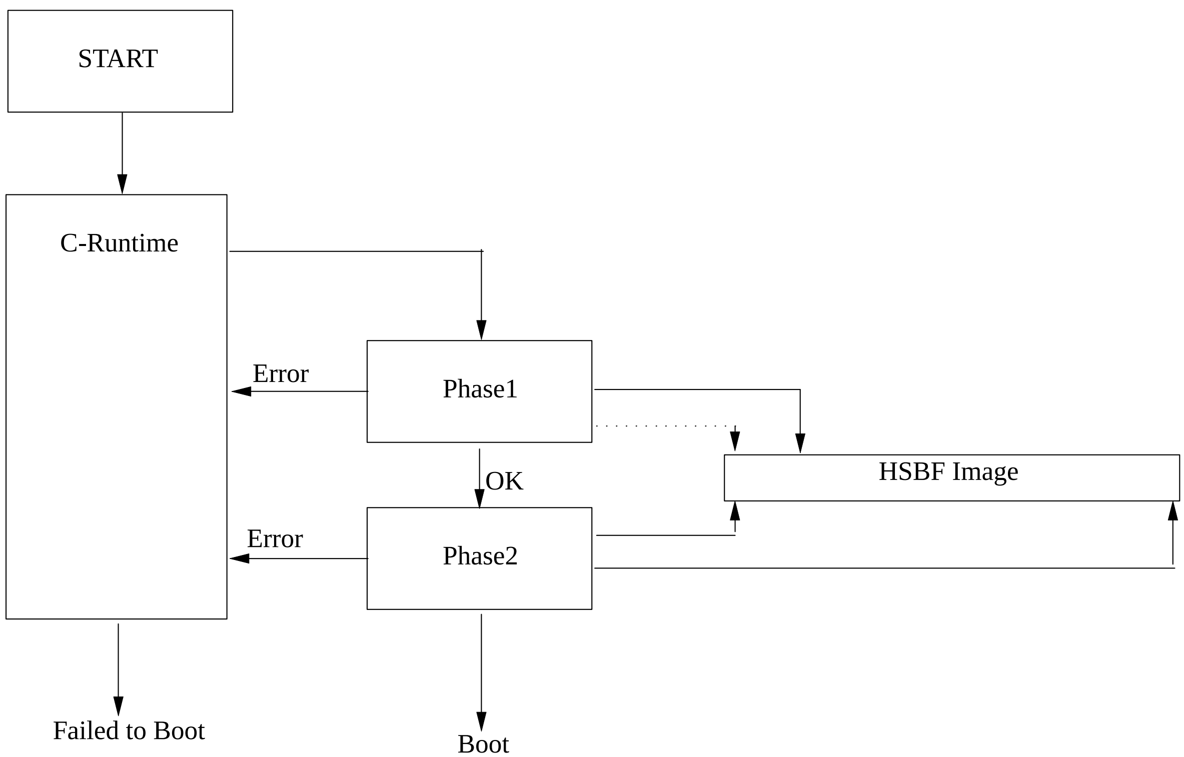
```
void bios_memory_mmu_setup(void)
```


Chapter 6 .

Module bios_high

Purpose

This module contains the boot logic that retrieves the boot image from external storage, verifies the content, process the verified objects found in the image. Finally, control is transferred to loaded boot object (Hypervisor).



HSBF Image

The Haspoc Boot Format corresponds to the Firmware Image Package (FIP-format) in the reference design for trusted boot provided by ARM. The format is simplified, but still offers the same cryptographic strength. Some entities such as service runtime could be verbatim migrated from FIP format into the HSBF container format.

Memory management

Initially the amount of memory is limited to static ram. To be able to process the HSBF image additional memory is required. Dependent of the platform, configuration of high capacity dynamic memory may require firmware to be obtained and passed to memory subsystem. The design therefore provides a mechanism that reads the HSBF image in two phases (pass1 and pass 2).

- Initial phase for memory configuration
- Final phase for boot

The design a based on dynamic memory management (free/malloc). For the initial phase static memory is used. When the dynamic memry is configured, the heap is assigned to this memory.
The start and size of dynamic memory being used are defined in the HSBF image, which allows customization of the address range for different application.

Interface

bios_high_runtime

This function assigns a small heap
residing in static ram and then invoke main

void bios_high_runtime(void)

_main

This function retrieves each object
in the HSBF image and then boot
Upon sucess it never returns

int _main(boot_start_sector, initial_read)

Name	Direction	Type	Purpose
boot_start_sector	in	int	first sector of HSBF image
initial_read	in	int	limited read in pass 1

Return Codes

- BIOS_HIGH_FAILED_RETRIEVE_ROOT_PK_HASH
- BIOS_HIGH_FAILED_INSTALL_ROOT_PK_HASH
- BIOS_HIGH_MALLOC_SRAM_FAILED
- BIOS_HIGH_FAILED_VERIFY_ROOTPK
- BIOS_HIGH_ROOTPK_NOT_FOUND
- BIOS_HIGH_FAILED_VERIFY_DRAM
- BIOS_HIGH_DRAM_NOT_FOUND
- BIOS_HIGH_MALLOC_DRAM_FAILED
- BIOS_HIGH_FAILED_VERIFY_ATF_BOOT
- BIOS_HIGH_FAILED_VERIFY_RAMDISK
- BIOS_HIGH_FAILED_VERIFY_ATF_PAYLOAD
- BIOS_HIGH_FAILED_VERIFY_ATF_RUNTIME
- BIOS_HIGH_FAILED_UNKNOWN_OBJECT
- BIOS_HIGH_UNKNOWN_SIZE_HSBF
- BIOS_HIGH_FAILED_NO_BOOT_OBJECT
- BIOS_HIGH_FAILED_ATF_BOOT
- BIOS_HIGH_BAD_STORAGE_READ
- BIOS_HIGH_SDRAM_SETUP_ERROR
- BIOS_HIGH_FAILED_INIT_STORAGE

Chapter 7 .

Module spdlib

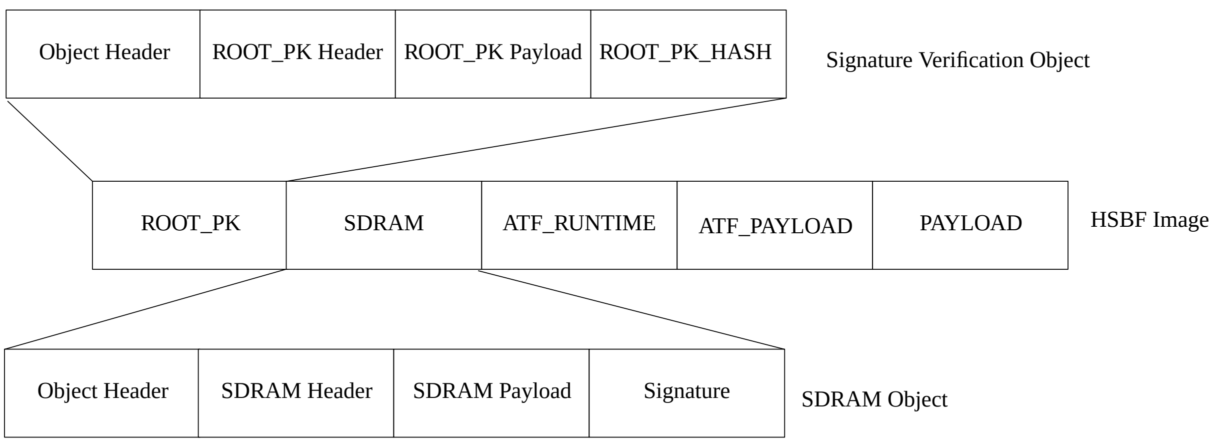
Purpose

This module provides format and support library for secure management of elements needed to boot the target.

Haspoc Secure Boot Format

The HSBF-image could be stored as a file, or directly written to consecutive region of fix length sectors. The image contains arbitrary number of concatenated objects, therefore the size of the entire image could be stored inside the image.

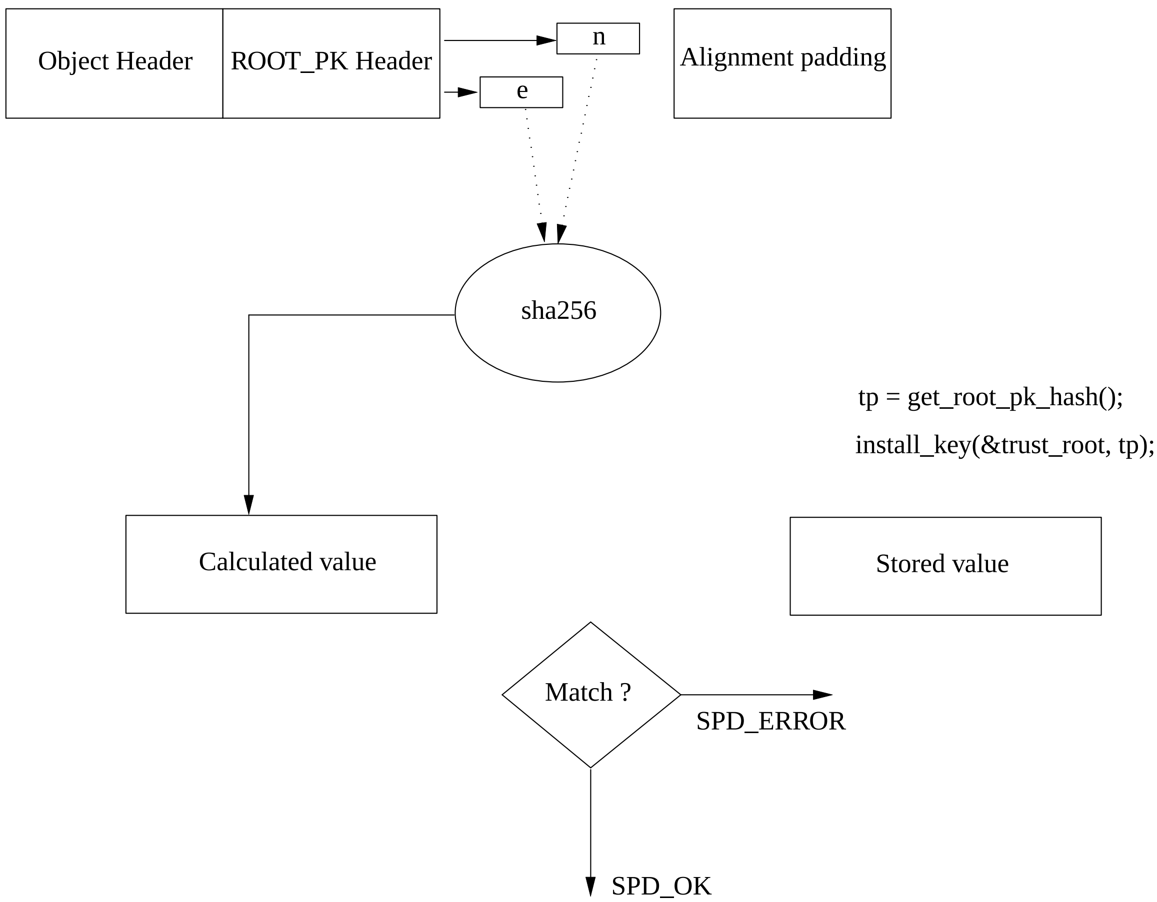
Each object is aligned to match 64 bit architectures, which enables faster processing of all object contained in the image.



For Trustzone enabled platforms, execution starts in a resource constrained environment, therefore the SDRAM object must be allocated in the beginning of the image.

Initial trust anchoring

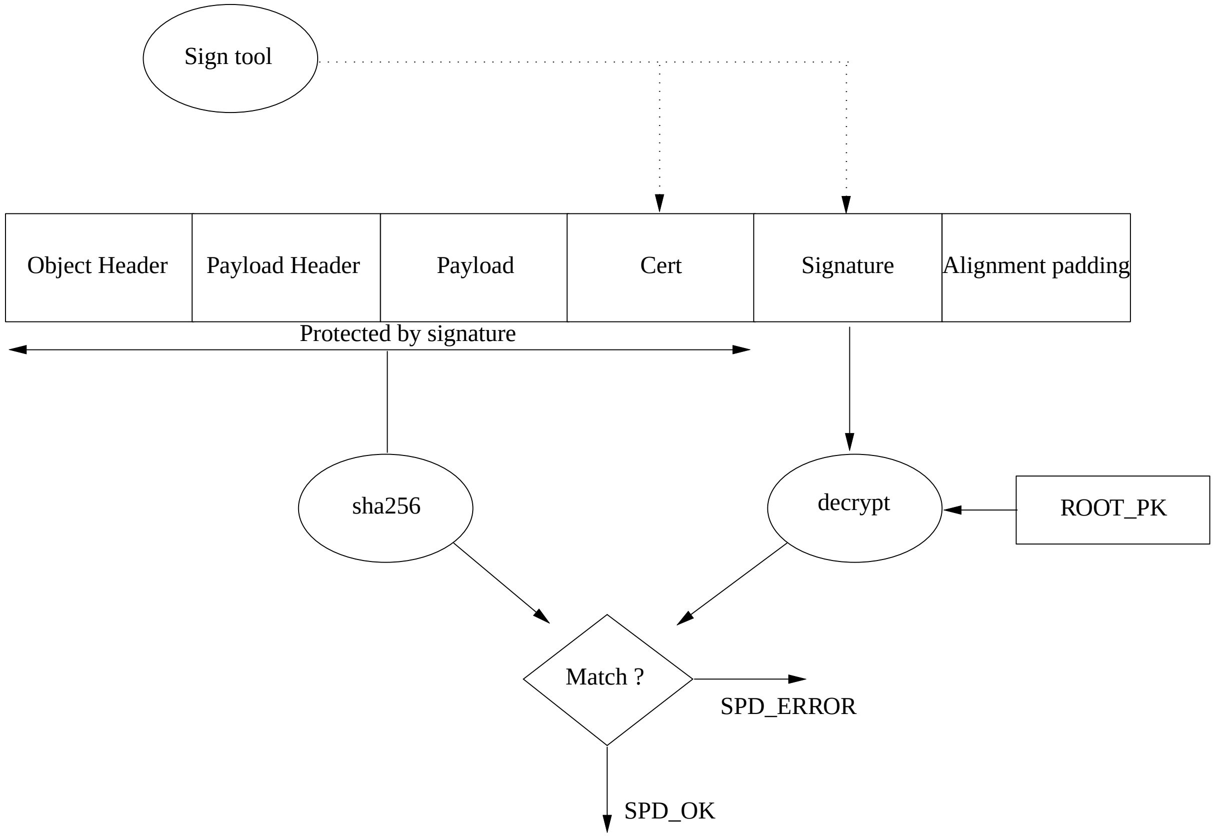
The trust anchor require a tamper resilient storage of the public key used for verification. Modern hardware provides fuse blow registers enabling key exchange without jeopardizing resilience. Due to the limited length of such registers, the hash value of the actual key is stored instead of the key itself.



Access to the hash value (ROOT_PK_HASH) of the public key (ROOT_PK) is abstracted by a function that enables adaptation to the actual hardware. When the public exponents are extracted from the encapsulating format, a hash is calculated and then compared with the stored value. Upon success the public key is installed as a new trust anchor. From now on objects will be verified against this key.

Object verification

Verification of digital signatures have been implemented as simple as possible. Involving X509 certificates involves expiration dates, revocation etc. The signer tool executes in a rich environment and support public key infrastructure, therefore the HSBF format provides a placeholder for the corresponding certificate. This enables future extensions where the certificate is used in the boot process.



Signature verification with asymmetric keys actually involves one hash calculation and one decryption involving the public key, as show in the picture above. The hash is calculated of all sensitive data, it is then compared with the decrypted corresponding hash generated by the signer tool. Upon success the two hash values are equal.

Interface

install_key

This function manages the trust anchor by installing either
the ROOT_PK_HASH or the asymmetric key ROOT_PK

int install_key(trust, sp)

Name	Direction	Type	Purpose
trust	in	struct sobject **	pointer to root
sp	in	struct sobject *	pointer to trust data

Return Codes

- SPD_OK
- SPD_UNSUPPORTED_ROOT_PK_TYPE
- SPD_INVALID_POINTER_HSBF_IMAGE

verify_element

This function verifies HSBF element in a sequence starting
at address passed as second parameter. Verification is performed
against the trust parameter.
Upon success the next pointer in the HSBF is updated to point to
next element , or NULL if no more element being found.

int verify_element(trust, sp)

Name	Direction	Type	Purpose
trust	in	struct sobject *	pointer to root
sp	in	struct sobject *	pointer to boot object

Return Codes

- SPD_OK
- SPD_ERROR
- SPD_UNSUPPORTED_HASH_TYPE
- SPD_HASH_LENGTH_MISMATCH
- SPD_UNSUPPORTED_SIGNATURE_TYPE
- SPD_MEMORY_ALLOCATION_FAILURE
- SPD_UNSUPPORTED_ROOT_PK_TYPE
- SPD_TRUST_EXPECTED_ROOT_PK
- SPD_EXPECTED_RSA
- SPD_INVALID_POINTER_HSBF_IMAGE
- SPD_HSBF_UNSUPPORTED_OBJECT

Structures

atf_payload

Member Name	Type	Description
start	uint64_t	Start of fip.bin relative start of this header.
bl33_load_address	uint64_t	Where to load bl33.
size	uint32_t	Size of payload image.

atf_runtime

Member Name	Type	Description
start	uint64_t	Start of fip.bin relative start of this header.
bl31_load_address	uint64_t	Where to load bl31.
size	uint32_t	Size of payload image.

boot_cfg

Member Name	Type	Description
tbd_1	uint32_t	To be defined.
tbd_2	uint32_t	To be defined.

boot_ldr_2

Member Name	Type	Description
start	uint64_t	Start of fip.bin relative start of this header.
bl31_load_address	uint64_t	Where to load bl31.
bl32_load_address	uint64_t	Where to load bl32.
bl33_load_address	uint64_t	Where to load bl33.
bl33_X0	uint64_t	X0 argument to bl33.
bl33_X1	uint64_t	X1 argument to bl33.
e_level	uint32_t	Exception level of bl33 [1-2].
size	uint32_t	Size of payload image.

devtree

Member Name	Type	Description
start	uint64_t	start of devtree image
load_address	uint64_t	Where to load device tree.
size	uint32_t	Size of device tree.

kernel

Member Name	Type	Description
start	uint64_t	start of kernel image
load_address	uint64_t	Where to load kernel.
size	uint32_t	Size of kernel.
X0	uint64_t	X0 argument.
X1	uint64_t	X1 argument.
e_level	uint32_t	Exception level of bl33 [1-2].

ramdisk

Member Name	Type	Description
start	uint64_t	start of ramdisk image
load_address	uint64_t	Where to load ramdisk.
size	uint32_t	Size of ramdisk.

root_pk

Member Name	Type	Description
type	uint32_t	Type of object.
tag_0	uint32_t	Type of first element.
value_0	uint64_t	Start of first element.
size_0	uint32_t	Size of first element.
tag_1	uint32_t	Type of second element.
value_1	uint64_t	Start of second element.
size_1	uint32_t	Size of second element.
id	uint8_t	Identifier.

root_pk_hash

Member Name	Type	Description
type	uint32_t	Type of hash.
size	uint32_t	Size of hash.
start	uint64_t	Start of hash.

sdram

Member Name	Type	Description
start	uint64_t	Start of image relative start of this header.
heap_start	uint64_t	Start of heap in SDRAM.
heap_size	uint32_t	Size of heap in SDRAM.
size	uint32_t	Size of image.
type	uint32_t	type
image_size	uint32_t	Total size of HSBF image.

sobject

Member Name	Type	Description
version	uint32_t	Version of format.
type	uint32_t	Object identifier.
cert_start	uint64_t	
sigstart	uint64_t	Start of signature.
padding	uint32_t	Memory alignment.
cert_len	uint32_t	
siglen	uint32_t	Length of signature.
sigtype	uint32_t	Signature type.

Chapter **8** .

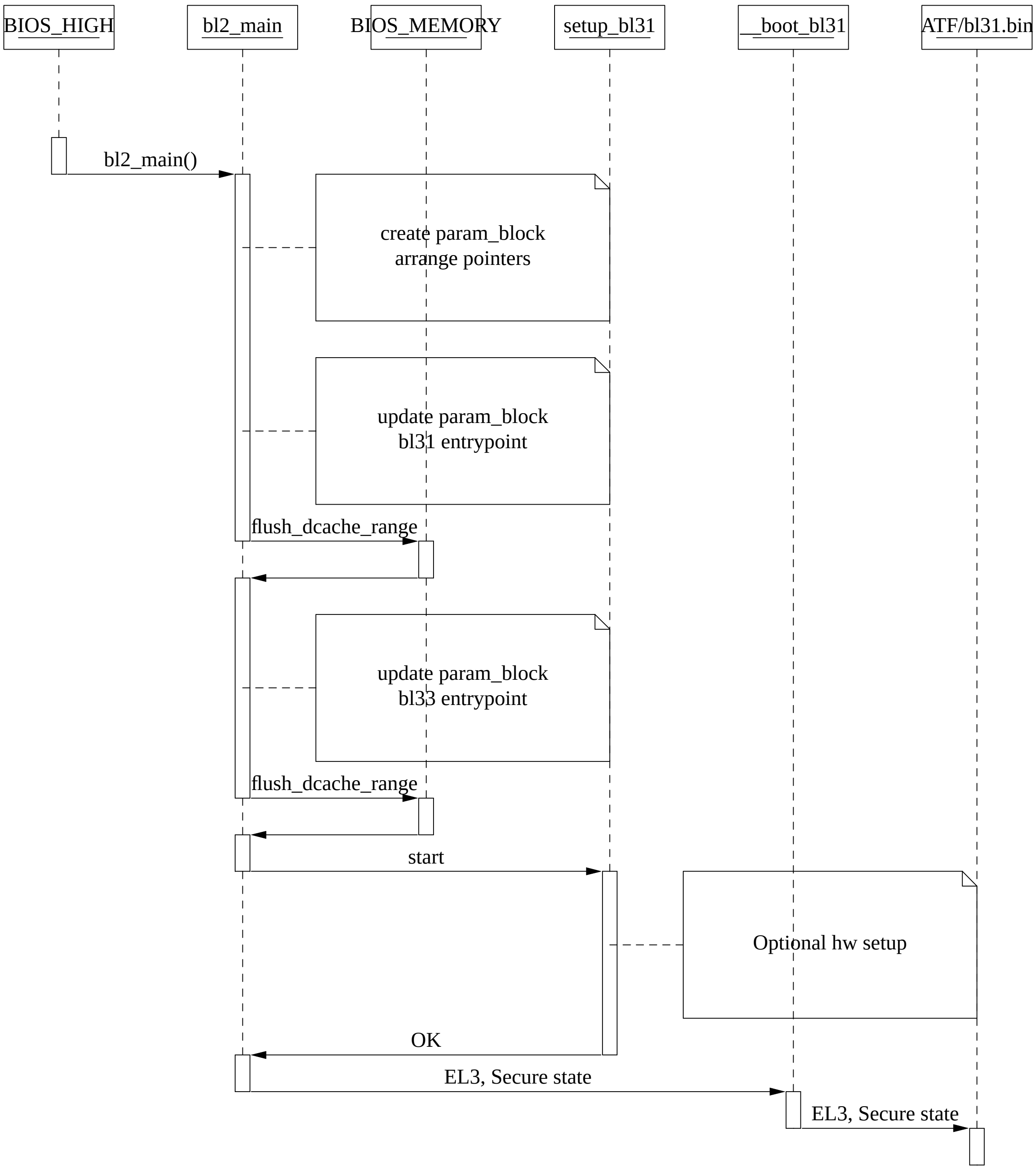
Module bios_fip

Purpose

This module provides compatibility with Arm Trusted Firmware modules, by support for the FIP (Firmware Image Package) format , and to generate the parameter block required to invoke external modules.

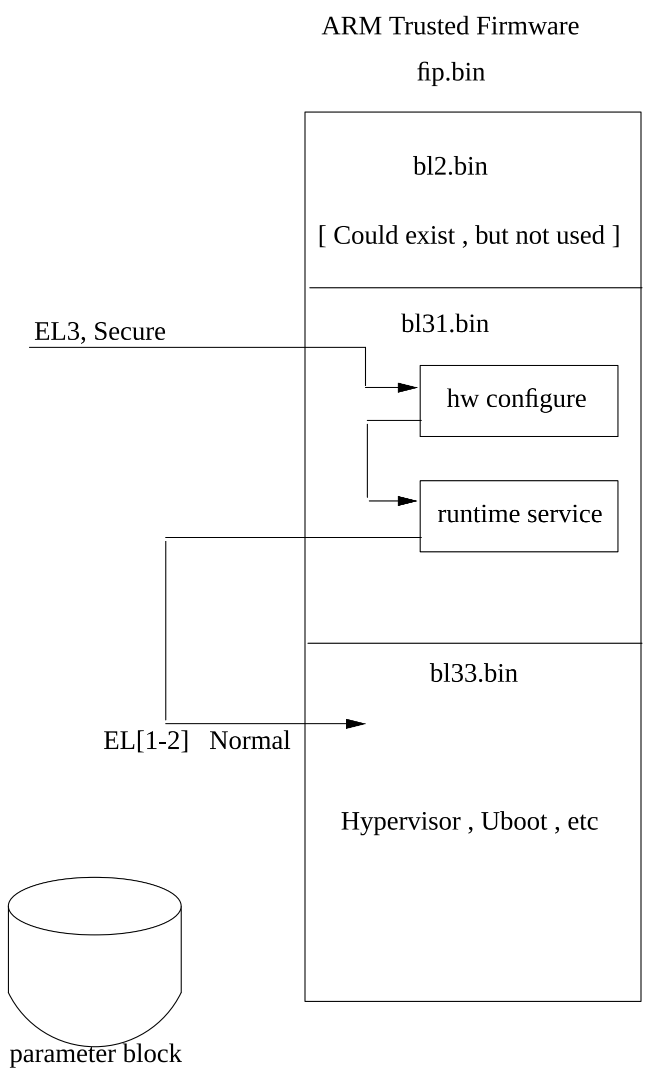
Arm Trusted Firmware Compliance

The bl2_main function reflects the functionality of the corresponding bl2.bin module in Arm Trusted Firmware.



The parameter area is written by this module and used by bl31.bin from Arm Trusted Firmware. The parameters are then updated, depending of the location and size of each image. Since cache is enabled, the memory system must be flushed to assure that images is written to memory.

The picture below illustrates how the runtime module (bl31.bin) from ARM Trusted Firmware is integrated into Secure Boot. The hardware configuration could be migrated into Secure boot into the optional function in this module. The call into runtime service installs the interrupt handlers reguired to process SMC calls used for power management. The final phase is the downshift in exception level, from EL3 to either EL2 or EL1, the actual value is specified in the HSBF image.



Interface

bl31_setup

This function allows migration of overlapping
setup from bl31 to Secure Boot

bl31_early_platform_setup()
bl31_plat_arch_setup()
bl31_arch_setup()
bl31_platform_setup()

int bl31_setup(void)

Return Codes

- FIP_OK
- FIP_ERROR

bl2main

This function prepares the parameter block needed
for boot compliance with ATF 1.1

The Header contains load addresses for each object,
for bl33 the exception level is specified in the header

The location of the parameter block is platform
dependent and defined by the linker script __PARAMS_BASE__.

int bl2main(e_level, bl31_load_address, bl31_size, bl33_load_address, bl33_size, bl33_X0, bl33_X1)

Name	Direction	Type	Purpose
e_level	in	uint32_t	exception level for bl33
bl31_load_address	in	uint64_t *	where to load bl31
bl31_size	in	int	size of atf runtime
bl33_load_address	in	uint64_t *	where to load bl33
bl33_size	in	int	size of atf payload
bl33_X0	in	uint64_t	first argument to bl33
bl33_X1	in	uint64_t	second argument to bl33

Return Never upon success

Return Codes

- FIP_ERROR_BOOT_FAILED
- FIP_ERROR_BL31_INVALID_LOAD_ADDRESS
- FIP_ERROR_BL33_INVALID_LOAD_ADDRESS
- FIP_ERROR_BL31_NOT_FOUND
- FIP_ERROR_BL33_NOT_FOUND

Structures

aapcs64_params

Member Name	Type	Description
arg0	uint64_t	X0.
arg1	uint64_t	X1.
arg2	uint64_t	X2.
arg3	uint64_t	X3.
arg4	uint64_t	X4.
arg5	uint64_t	X5.
arg6	uint64_t	X6.
arg7	uint64_t	X7.

bl2_to_bl31_params_mem

Member Name	Type	Description
bl31_params	bl31_params_t	BL31 Call Interface.
bl31_image_info	image_info_t	Image info about Runtime.
bl32_image_info	image_info_t	Image info about Trusted OS.
bl33_image_info	image_info_t	Image info about Bootloader.
bl33_ep_info	entry_point_info_t	Execution info about Bootloader.
bl32_ep_info	entry_point_info_t	Execution info about Trusted OS.
bl31_ep_info	entry_point_info_t	Execution info about Runtime.

bl31_params

Member Name	Type	Description
h	param_header_t	version
bl31_image_info	image_info_t*	Image info pointer to bl31.
bl32_ep_info	entry_point_info_t*	Execution info pointer to bl31.
bl32_image_info	image_info_t*	Image info pointer to bl32.
bl33_ep_info	entry_point_info_t*	Execution info pointer to bl33.
bl33_image_info	image_info_t*	Image info pointer to bl33.

entry_point_info

Member Name	Type	Description
h	param_header_t	version
pc	uint64_t	start address for link register
spsr	uint32_t	Processor state.
args	aapcs64_params_t	X0 - X7 parameters.

image_info

Member Name	Type	Description
h	param_header_t	version
image_base	uint64_t	location of image
image_size	uint32_t	bytes read from image file

param_header

Member Name	Type	Description
type	uint8_t	type of the structure
version	uint8_t	version of this structure
size	uint16_t	size of this structure in bytes
attr	uint32_t	attributes: unused bits SBZ

Chapter 9 .

Module bios_trust_anchor

Purpose

The integrity control of the platform must be an integral part of the hardware to be resilient against tampering. This module provides a interface that allows the retrieval of ROOT_PK_HASH to be customized.

Example

The hash value stored in the target is calculated by concatenating the RSA public key exponents.

```
/*
	Input build.pem
	Generated by hsbf_signer

	Subject:		/CN=t2data.com/emailAddress=ca@t2data.com/O=T2Data/OU=pki/ST=Stockholm/C=SE
	size of n = 256 , e = 3

	The content of root_pk_hash_value array,
	is calculated by hashing the e and n  exponent for a RSA key
	Prior to use of the actual ROOT_PK key the hash must match

	The trust chain depends on how the root_pk_hash_value is stored
*/

#define  ROOT_PK_HASH_TYPE SHA256_BIN

#define  ROOT_PK_HASH_SIZE 32

static const char root_pk_hash_value[] = {
	0x52,0x24,0xa5,0x41,0xc6,0x56,0x4e,0x89,
	0x66,0x6c,0x2e,0x10,0xbc,0x77,0x2,0x0,
	0xbe,0x36,0x51,0x4c,0xaa,0x65,0x40,0x21,
	0xc0,0x7a,0x50,0x99,0x79,0xa6,0x53,0x4f
	};
```

Interface

get_root_pk_hash

Returns pointer to structure containing
hashed value of public key.

struct sobject* get_root_pk_hash(void)
Return pointer to sobject or NULL

Chapter 10 .

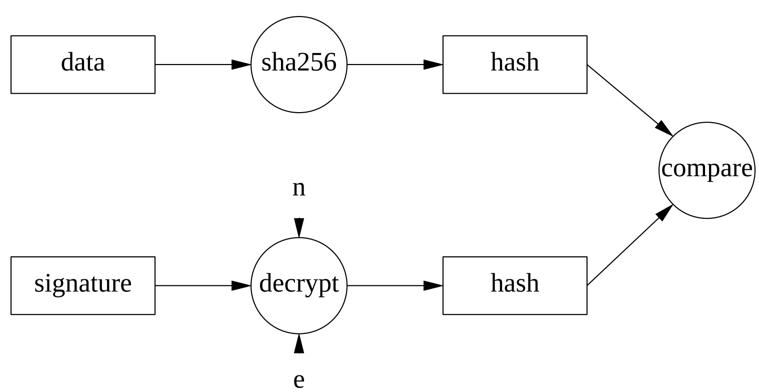
Module secure_bios

Purpose

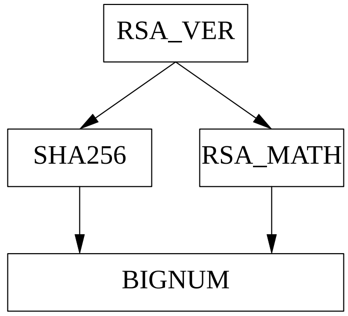
This module provides cryptographic functions needed to verify signatures and hashes.

Overview

The input to verification consist of the data itself and the signature. The signature is generated by the issuer of the data and contain the encrypted hash value of the data. The issuer uses the private key for encryption.
The verification process compares two hash values. The first hash is calculated by the receiver using the same hash algorithm as the issuer (SHA-256) on the raw data, and the second hash is generated by decrypting the signature. If the two hash values match, then the signature is considered valid.
For X509 the RSA public key exponent (e) value is always the same and could be implemented as a constant in the algorithm.



The design of this module includes the functions below



Interface

sha_256

Uses the SHA-256 algorithm on the input data. The output hash will have a length of 65 bytes, 64 bytes hex-values and a null-terminator.

void sha_256(p, len, output)

Name	Direction	Type	Purpose
p	in	void *	Pointer to data to be hashed
len	in	int	Length of data
output	out	unsigned char *	binary sha256 hash

rsa_signature_check

This function uses the RSA decrypt- and SHA-1 hash- functions to decrypt the signature, hash the data and then compare them to see if the signature is valid or not. It requires the RSA public key exponent (e) and the RSA modulus (n). The e value is common for keys used in X509 certificates and could therefore be excluded for key storage.

int rsa_signature_check(e, e_len, n, n_len, sig, sig_len, data, data_len)

Name	Direction	Type	Purpose
e	in	unsigned char *	RSA public key exponent
e_len	in	unsigned short	Length of RSA public key exponent
n	in	unsigned char *	RSA modulus
n_len	in	unsigned short	Length of RSA modulus
sig	in	unsigned char *	Signature to be decrypted
sig_len	in	unsigned short	Length of signature to be decrypted
data	in	unsigned char *	Data to be hashed
data_len	in	unsigned long	Length of data to be hashed

Return 1 if signature is valid and 0 if invalid

Purpose

Provide portable runtime functions similar to a Unix C-library. For test purposes the same functionality could be implemented on a different architecture, allowing module tests during software build.

Memory management

The design of Secure Boot depends on dynamic memory management, based on the malloc/free paradigm.
Since Boot by nature is both transient and predictable, the design could be very simple. Instead of re-use memory given back by a call to free, the design continue to use new unused memory instead.

Performance issues

The generic and portable function NMemCopy is not optimal for performance, since it copies only one byte. By copying multiple bytes a significant performance boost could be achieved.

- plf_memcpy16
- FMemCopy

Interface

NMemPool

When memory management is used in realtime applications malloc and free often operates on a predefined memory area that is managed very efficient and fast.

int NMemPool(start, size, test)

Name	Direction	Type	Purpose
start	in	char *	First address of memory region
size	in	size_t	in bytes of memory regions
test	in	int	enabled if true

Return Codes

- 1
- 0

Module plf_bios_arm

NMemAlloc_align

This function allocates a new memory buffer, where all bytes are initialized to 0. The align argument must be a multiple of 2

void* NMemAlloc_align(bytes, align)

Name	Direction	Type	Purpose
bytes	in	size_t	The number of bytes to allocate.
align	in	size_t	to place the memory

Return pointer to allocated buffer.

NMemCopy

This functon creates a copy of a memory buffer.

void* NMemCopy(dest, src, count)

Name	Direction	Type	Purpose
dest	out	void *	Pointer to destination buffer.
src	in	void *	Pointer to source buffer.
count	in	size_t	Number of bytes to copy.

Return pointer to destination buffer.

NMemAlloc

This function allocates a new memory buffer, where all bytes are initialized to 0.

void* NMemAlloc(bytes)

Name	Direction	Type	Purpose
bytes	in	size_t	The number of bytes to allocate.

Return pointer to allocated buffer.

Secure boot is a suite that integrate several modules, where some modules is generic and reusable among different underlying hardware. Other modules configuring hardware are specific. The output module is optional , useful in the development phase , but adds complexity to evaluation and testing.
The dependency table for the TSFI therefore is based on information from the build process and reflects the actual usage of the interfaces provided by modules. The documentation for each module cover all interfaces , therefore there could be discrepancy between the available interfaces , and interfaces actual used. The build process only selects interface actually used.

Design issues and TSFI

TSFI_POWER_COLD

This external interface is triggered by reset or power on.

- Generic configuration of TOE
- Platform specific configuration of TOE
- Transfer control to boot application (TSFI_STORAGE / _main())

TSFI_STORAGE

This external interface is triggered when a cold boot is complete (TSFI_POWER_COLD).

- Initialize storage device
- Retrieve image from storage device
- Verify integrity of objects embedded in retrieved image
- Copy verified objects into memory
- Transfer control to boot object (ATF bl31)

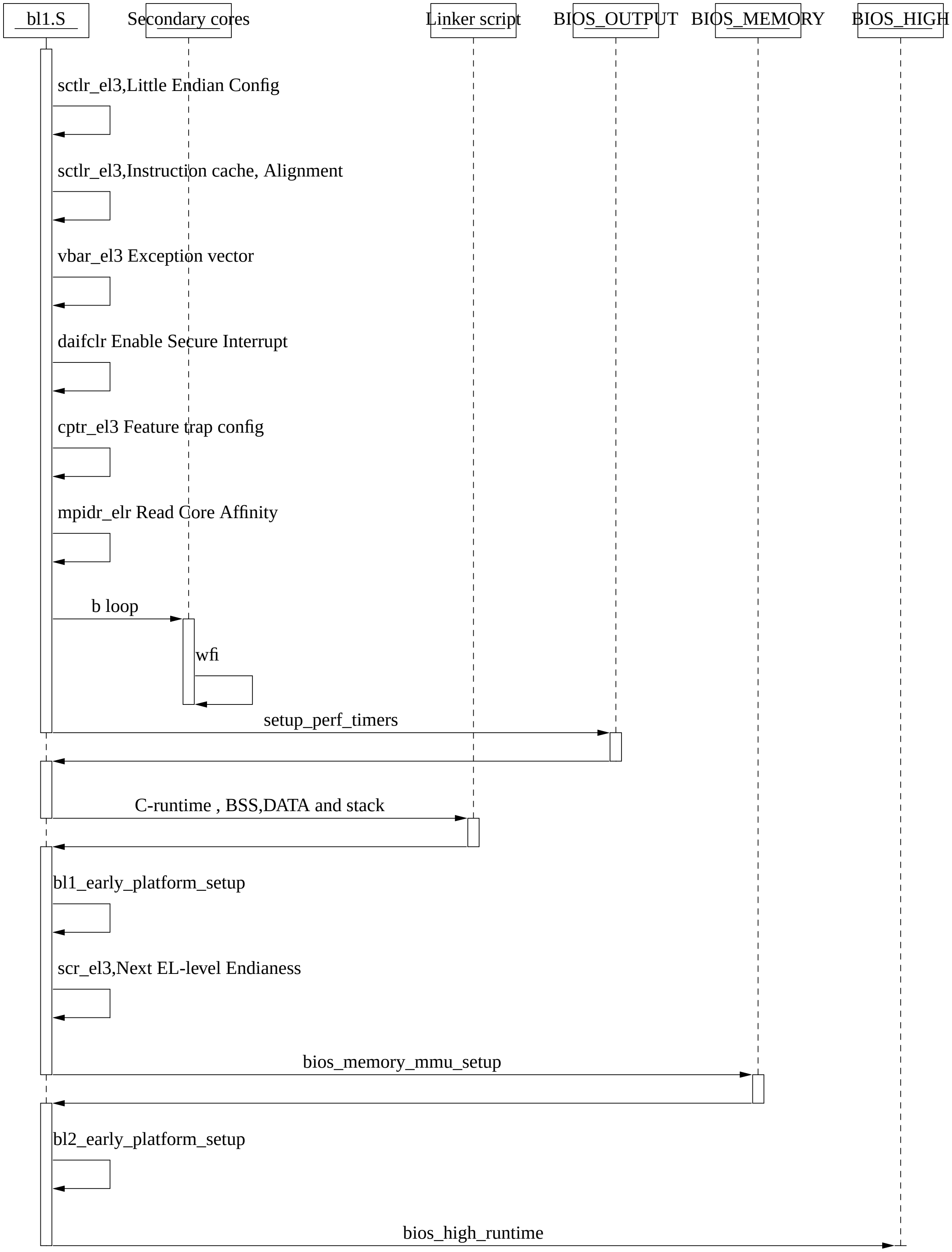
Interaction

The control flow is represented by sequence diagram.

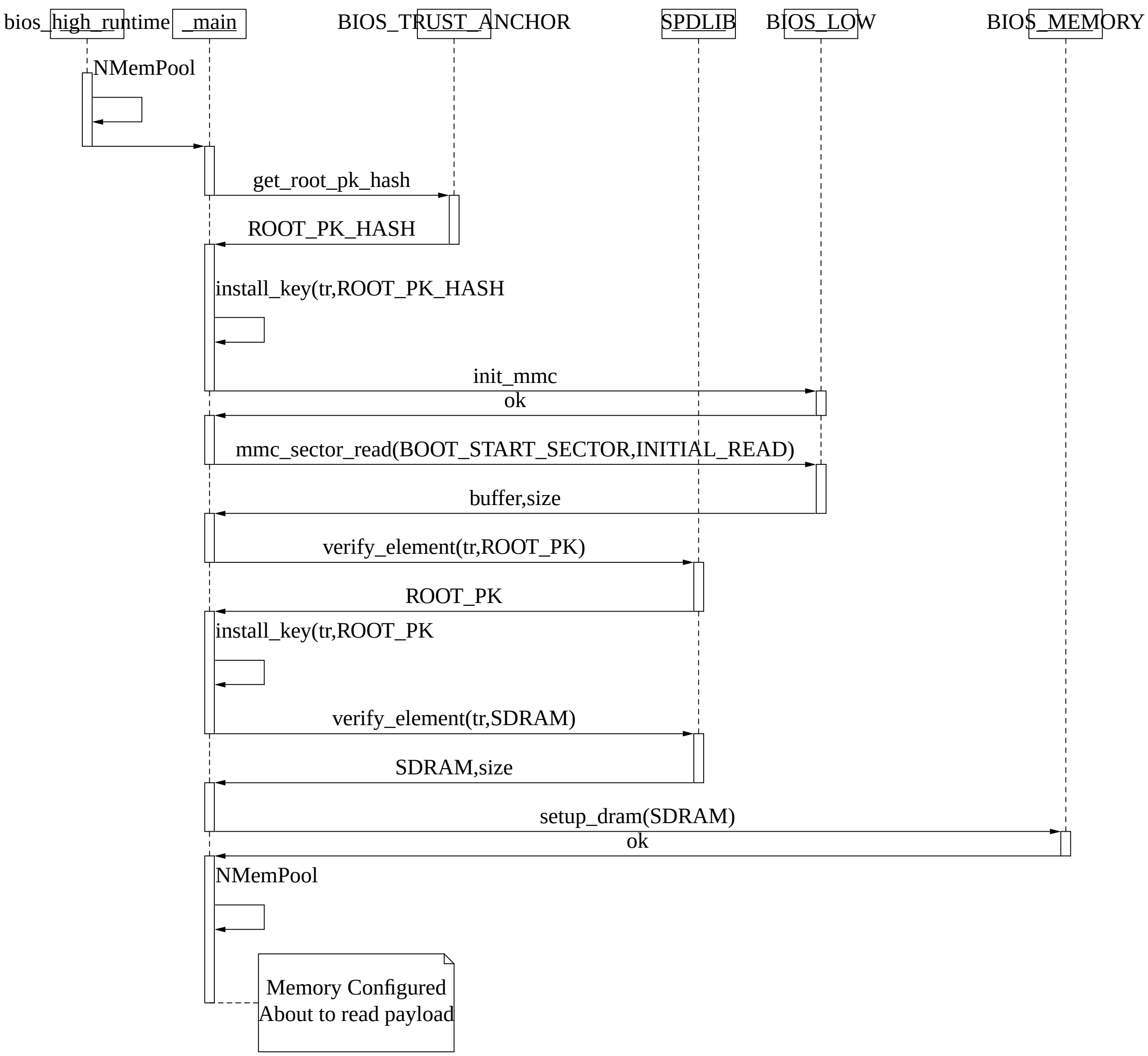
- bios_lowhardware specific
- bios_highhardware agnostic

Bios low

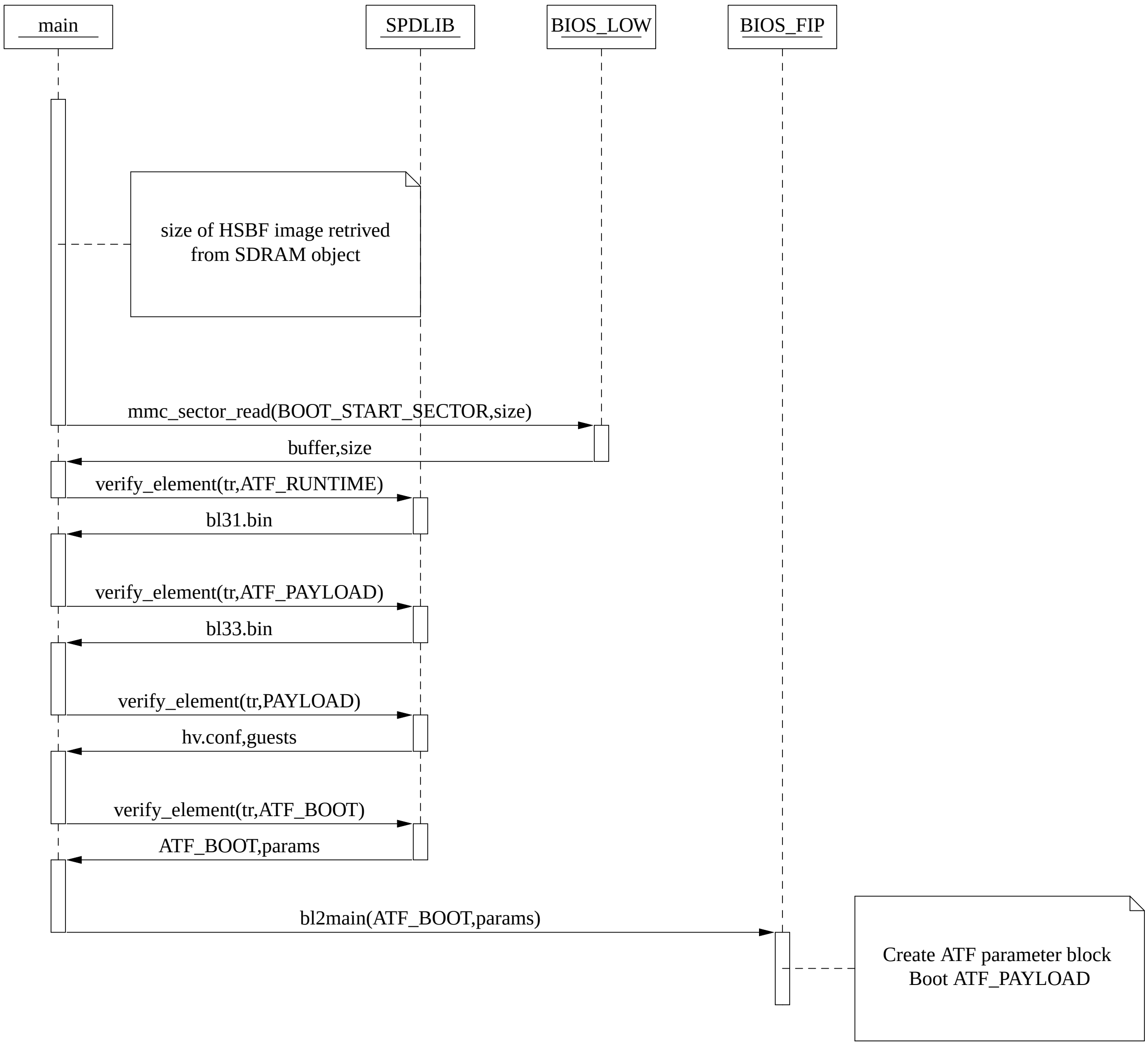
-2-


bl1.S
Secondary cores
Linker script
BIOS_OUTPUT
BIOS_MEMORY
BIOS_HIGH
sctlr_el3,Little Endian Config
sctlr_el3,Instruction cache, Alignment
vbar_el3 Exception vector
daifclr Enable Secure Interrupt
cptr_el3 Feature trap config
mpidr_elr Read Core Affinity
b loop
wfi
setup_perf_timers
C-runtime , BSS,DATA and stack
bl1_early_platform_setup
scr_el3,Next EL-level Endianess
bios_memory_mmu_setup
bl2_early_platform_setup
bios_high_runtime

Bios high sequence 1

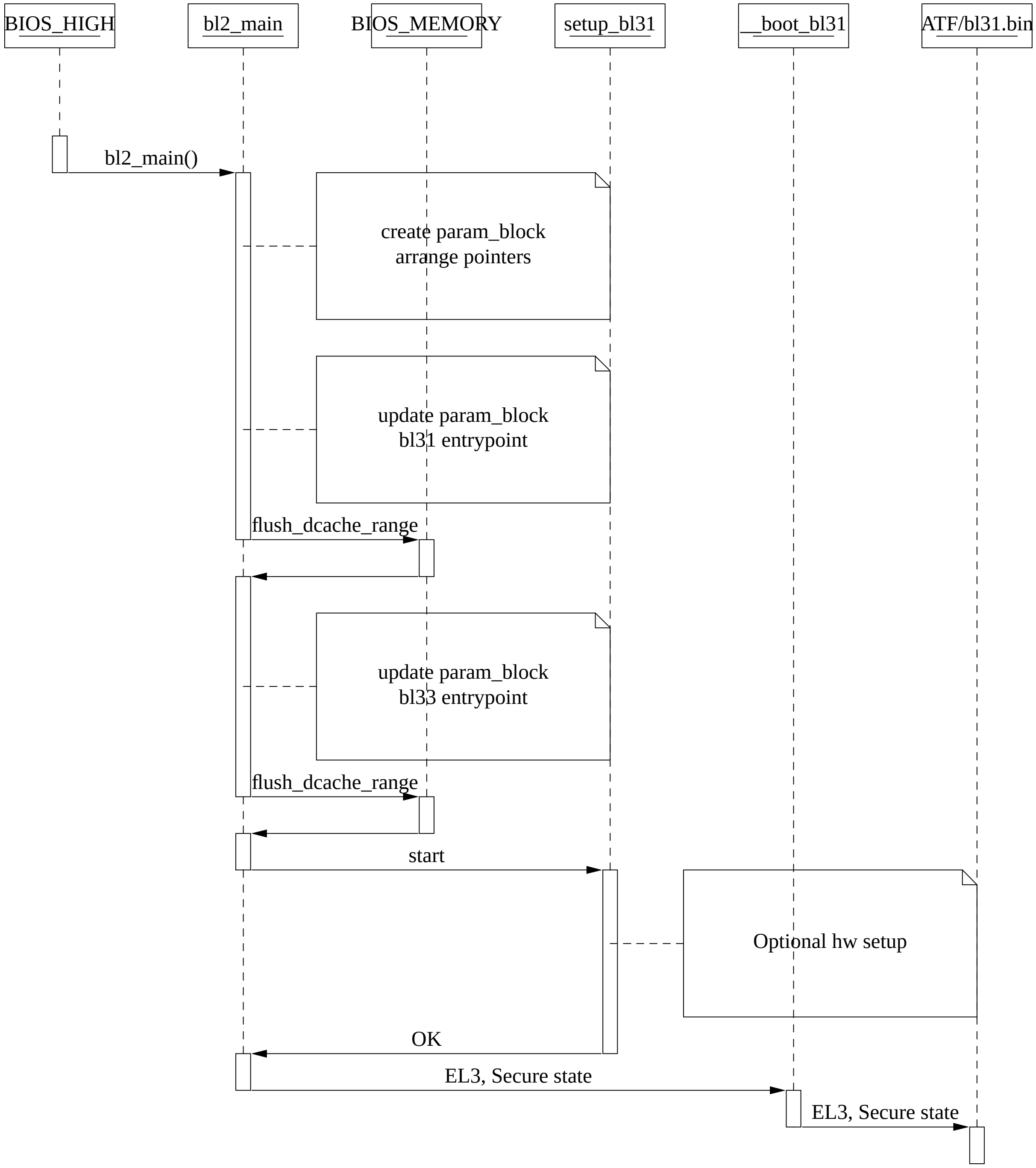


Bios high sequence 2



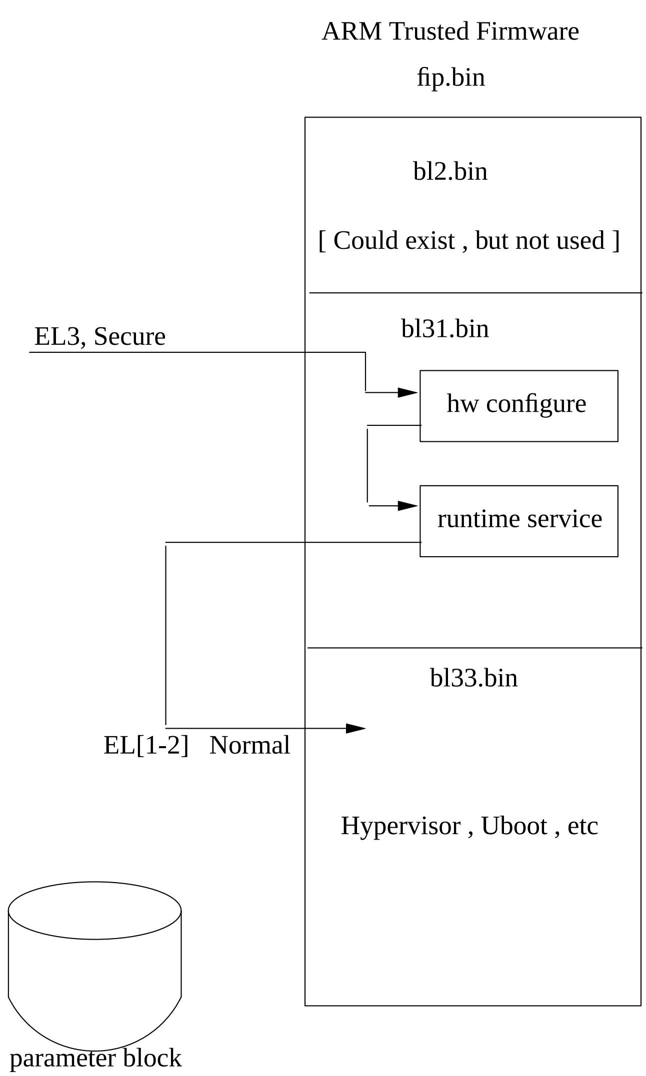
bl2_main

This function enables compability with Arm Trusted Firmware.


BIOS_HIGH
bl2_main
BIOS_MEMORY
setup_bl31
__boot_bl31
ATF/bl31.bin
bl2_main()
create param_block
arrange pointers
update param_block
bl31 entrypoint
flush_dcache_range
update param_block
bl33 entrypoint
flush_dcache_range
start
Optional hw setup
OK
EL3, Secure state
EL3, Secure state

ATF bl31

Note: This function is NOT part of Secure boot,



TSFI_STORAGE

bios_high

Relation	Module	Interface
Depend	Module	Interface
	bios_trust_anchor	`get_root_pk_hash`
	bios_fip	`bl2main`
	spdlib	`verify_element` `install_key`

	plf_bios_arm	`FMemCopy`
		`NMemPool`
		`NMemAlloc_align`
		`plf_memcpy16`

	bios_output	`printf`
	bios_memory	`setup_sdram`
	bios_low	`sector_read` `init_mmc`
Provide	Module	Interface
	bios_low	`bios_high_runtime`

TSFI_POWERON_COLD

bios_low

Relation	Interface	
	Module	Interface
Depend	bios_memory	dcsw_op_all dcsw_op_level1 dcsw_op_level2 bios_memory_mmu_setup
	plf_bios_arm	NMemCopy NMemSet
	bios_output	printf asm_print_str console_init asm_print_hex bios_output_setup_perf_timers
	bios_high	bios_high_runtime
	bios_low	hi6220_timer_init gpio_register_device gpio_get_value memcpy16 hi6553_read_8 bl1_early_platform_setup cci_init cci_enable_cluster_coherency hi6220_pll_mmc_init gpio_set_value gpio_direction_output init_acpu_dvfs udelay plat_reset_handler zeromem16 gpio_direction_input bl1_exceptions hi6220_pll_init hi6553_write_8 bl2_early_platform_setup mdelay plat_report_exception

bios_low

Relation	Interface	
	Module	Interface
Provide	bios_fip	`disable_mmu_icache_el3`

	bios_memory	`hisi_mcu_load_image`
		`hisi_mcu_start_run`
		`hisi_mcu_enable_sram`

	bios_high	`init_mmc` `sector_read`