MAX32550 Secure ROM User Guide

Ref: UG25H04
Rev H

MAXIM INTEGRATED PROPRIETARY – CONFIDENTIALITY

Revision History

Rev A	2013-Nov-28	1st release
Rev B	2014-Sep-15	2nd release
Rev C	2014-Sep-15	3rd release
Rev D	2014-Sep-19	4th release
Rev E	2014-Nov-6	5th release
Rev F	2014-Nov-17	6th release
Rev G	2015-Jul-01	7th release
Rev H	2015-Oct-13	8th release: key wipe description

Disclaimer

To our valued customers

We constantly strive to improve the quality of all our products and documentation. We have spent an exceptional amount of time to ensure that this document is correct. However, we realize that we may have missed a few things. If you find any information that is missing or appears in error, please send us an email via www.maximintegrated.com/support. We appreciate your assistance in making this a better document.

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Maxim Integrated Office.

Maxim Integrated technologies may only be used in life-support devices or systems with the express written approval of Maxim Integrated, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Table of Contents

Revision History **2**
Table of Contents **3**
List of Figures **5**
References **6**
Glossary **7**

List of Figures

References

[1] DS25H02: MAX32550 datasheet. Rev C. Jan-2015.

[2] SPEC25B02. MAX32550 SCP specifications. Rev A. Sep-2013

[3] UG25T01: MAX32550 Secure ROM package readme. Rev C. Sep-2014

[4] UG25K01: MAX32550 EVKit Software User Guide. Rev E. May-2014

[5] UG98S01: MPC, Production Center User Guide. Rev B. Jan-2015.

[6] Digital signature on Wikipedia: http://en.wikipedia.org/wiki/Digital_signature

[7] NIST FIPS 186-4: Digital Signature Standard. Jul-2013

[8] NIST FIPS180-2. Secure Hash Standard. Aug-2002

[9] PCI PTS Modular Security Requirements. v4.0. Feb-2013.

[10] UG25H05: MAX32550 Secure SOC User's Guide. Rev D May-2015.

Glossary

CRK	Customer ECDSA key
ECDSA	Elliptic Curve Digital Signature Algorithm
MRK	Maxim Root Key
OTP	One Time Programmable
RFU	Reserved for future use
ROM	Read Only Memory
RSA	Rivest Shamir Adleman
SLA	Second Level Application
SOC	System On Chip

1 INTRODUCTION

The MAX32550 is a secure SoC (see [1]). This document describes which services its Secure ROM code provide and how it simplifies the security requirements expressed in the most demanding security standards (e.g. see [9]).
It also presents the different boot scenarios and all mechanisms especially designed for PCI-PTS compliance.

1.1 Notations

Hexadecimal numbers are represented with a `0x` in front of them and in `Courier New`.

Example: `0xF8DF`

Decimal numbers are indicated by a 10 in index.

Example: 123_{10}

1.2 Endianness

The chosen convention is “big endian”. Hexadecimal numbers are presented most significant byte first and most significant bit first. It means the 32-bit number `0x123456` is 1193046_{10}.

2 Description

The MAX32550 secure SoC embeds a 64-KB ROM on its die.

The main objective for this ROM code is to guarantee the chain of trust, from reset to customer first application.

At reset, the chip automatically checks the integrity of this ROM code and jumps to the beginning of this ROM to start executing its code.
This ROM code is able to:

- Securely manage the chip life-cycle,
- Securely program the embedded NOR flash,
- Securely program the embedded OTP memory,
- Securely starts applications from embedded NOR flash
- Securely load and run test programs, using a flexible mechanism, using applets loaded in internal RAM,

2.1 Secure life-cycle

The cryptographic scheme used by the ROM code relies on ECDSA with curve *P-256*, using the FIPS 186-4 ECDSA standard (see [7]). All instances of "securely" below means "authenticated by a digital signature".

- By default, the chip is secure by the ECC P-256 Maxim Root Key, its public part being hard-coded in the ROM code.
- For unlocking chips, the customer has to generate his own ECC key pair, the Customer Root Key, CRK, and to securely program the public part in the chip OTP (the same key for all chips).
- Once this is done, the chip control is fully given to the customer and controlled by his CRK key.
- The customer may decide to kill the product (i.e. not responding any more), by securely sending a specific command to the ROM code.

2.2 Secure update

The ROM code proposes a service of embedded flash and OTP secure update
This secure update protocol is also able to program the MAX32550 internal OTP, used for memories and security configuration.
Thanks to a very powerful mechanism of applet loading, this secure update is also able to load and run small programs in internal RAM.
The links available for the secure download are serial port and USB link.

2.3 Secure boot

The ROM code, depending on the OTP configuration, loads, authenticates and runs the 2nd level application, i.e. the first customer application to be run after the ROM code has ended. The digital signature verification guarantees that no illicit application can be run from the MAX32550 secure SoC.

3 Tools and support

Maxim provides host tools (for Windows and Linux) able to build, sign and download data towards the MAX32550 chip, through the secure update mechanisms.
These tools include:

- tools for development download, using (non secure) development keys,
  - A secure ROM package, guiding users for their first use with a MAX32550, is available, using these tools,
- a 2nd level application development framework, with examples, based on the COBRA framework (see [4]),
- the MPC, a tool for production download, (see [5]),
- tools connected to a PKCS#11 HSM guaranteeing PCI-PTS-compliant CRK handling.

4 Boot scenarios

This chapter presents the ways customers applications can start on MAX32550, using the secure boot/secure loader ROM code.

Note: For more information on the configuration, see **Error! Reference source not found.**. For the secure protocol, see [2].

4.1 Secure boot

The secure boot is stored in the internal ROM memory of the secure SoC. This is the code that is always run first when the SoC starts up. This initiates the chain of trust that applications execution relies on: the secure boot guarantees to the applications the platform they are running on is trustworthy and is the one expected to be; the secure boot also guarantees that the applications running on the platform are only the authorized ones, so only trusted applications can run on the platform.

The security the secure boot implements to provide trust is based on standard public-key cryptography. The secure boot uses customer public key to verify the applications and to authenticate the updates (see section 5.1).

Objective of the secure boot is to provide high security strength to the platforms while keeping flexibility and ability to adapt to various customer needs. In that perspective, the secure boot has multiple mechanisms.

The first mechanism is, the verification before launch of user applications from flash memory: this verification guarantees that only authenticated applications can run on the platform.

The second mechanism is the secure loading in the Secure SoC internal RAM through the Secure Communication Protocol, SCP.

Note: the wording “application”, or 2nd level application (SLA), covers any kind of firmware binary that is run immediately after the secure boot. This application may be a 2nd level loader, a demo standalone application or even a full OS. It mainly depends on customer needs and memories configuration.

4.2 Customer key

The secure boot performs authentications using ECDSA P-256 algorithm (see [7]), with 256-bit ECC keypair and SHA-256 secure hash function (see [7]).

4.3 Secure scenario for MAX32550

As described above, the MAX32550 Secure boot starts in secure conditions and performs security controls (including cryptographic self-tests) before verifying the application stored in embedded flash memory. An usual RTOS-oriented software architecture can be built.

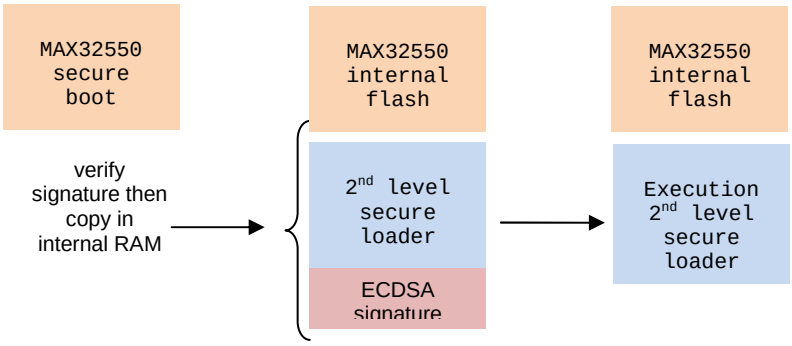


Figure 1 MAX32550 chain of trust

Note 1: the scenario described here shows that the application is run from the flash (XIP: "eXecute In Place").
Note 2: Maxim Integrated provides solutions such as toolchain/IDE framework, cryptographic library, software examples, smart card library for application development.

4.4 Application images

The 1MB flash is made of two 512-KB banks. When `SCON.FLASH_PAGE_FLIP=0`, for these 2 banks, logical and physical layouts match. (bank1 `0x10000000 – 0x1007ffff`, bank2 `0x10080000 – 0x100fffff`). Please refer to [10], §4.3.1, p34.

Note1: The application image has a specific format described in Figure 2.

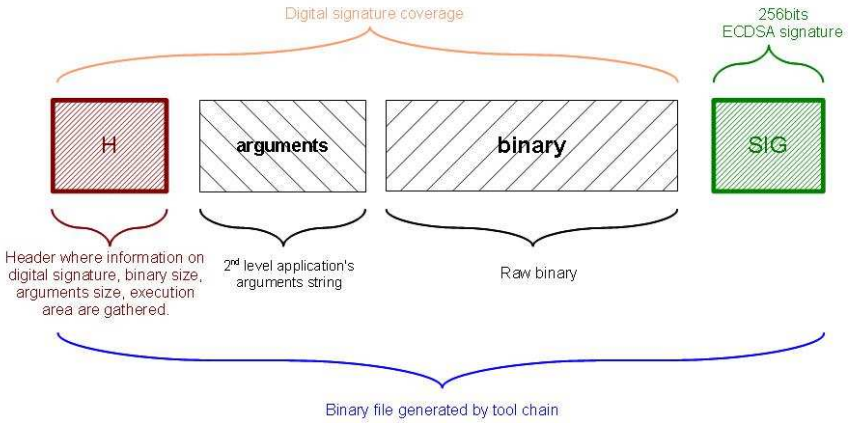


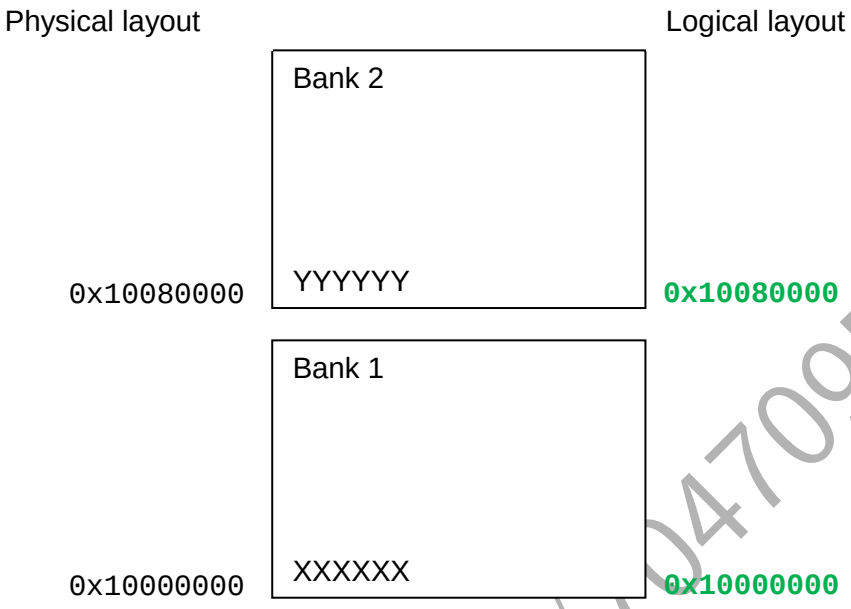
Figure 2 application format

- Secure header section
  - Synchronization pattern $_{(64bits)}$: `0x4447444557534948`.

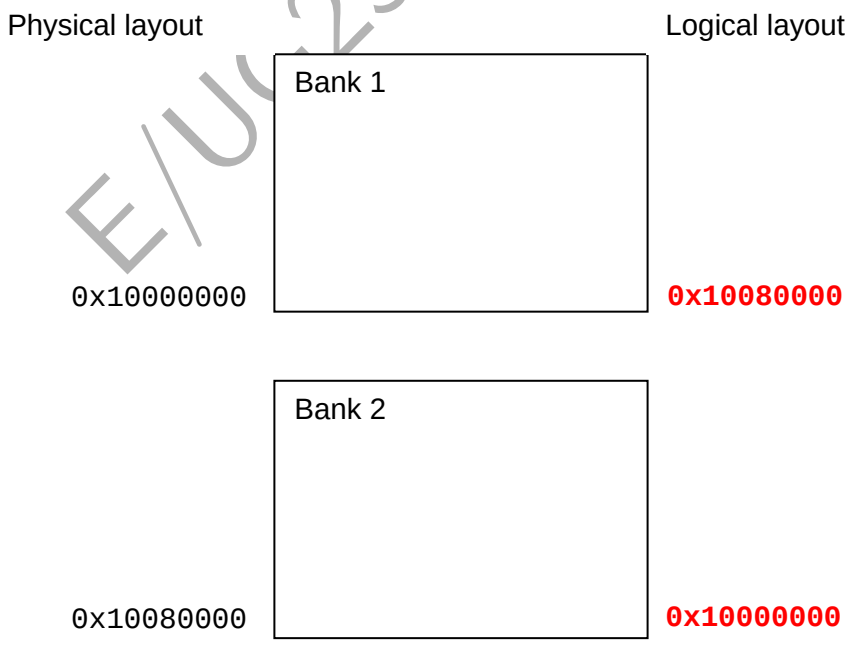
  - Format Version Number $_{(32bits)}$. Version of the binary format that shall match with what is expected by the ROM code (incremented for each new release of the ROM code). Since MAX32550-A3, ROM Code considers a "*reference version*" that could be "*older*" than current ROM Code one. This is the base version from which SLA has to refer (at minimum) to be granted to be run.
  - Raw binary load address $_{(32bits)}$. This address is where binary is going to be copied to.
  - Raw binary length $_{(32bits)}$. This length is the one of **binary itself**, *not binary section* !!!
  - Raw binary jump address $_{(32bits)}$. This address is where to start execution.
  - 2nd level application's arguments size $_{(32bits)}$. This is the size of arguments string.
  - Application version number $_{(32bits)}$: this number is used for comparison between the two potential headers found in the two banks. Please refer to §6.1.5 for "**Load & Launch**" sequence.
- Arguments section
  - It's where application's arguments are located.
- Application binary section
  - It's where application's raw binary is located.
- Signature section
  - This section holds the 512bits ECDSA signature of binary. This digital signature is generated on Host side. ROM Code role is to verify it. If it matches, execution is granted and ROM code will jump into binary entry point address. If signature does not match, ROM code turns platform into "*shutdown mode*". Digital signature is processed from synchronization pattern $_{(included)}$ to end of raw binary.

- The reset status is `SCON.FLASH_PAGE_FLIP =0`, so the ROM code sees the flash as 1MB flash (no switch)
- Flash addresses used SCP commands consider the whole flash (`SCON.FLASH_PAGE_FLIP =0`)
- The application secure header contains an absolute jump address, `jump-address`.
- Please refer to §6.1.5 for "**Load & Launch**" sequence $_{(SLA\ boot)}$.

SCON.FLASH_PAGE_FLIP=0 (default state, SCP state)



SCON.FLASH_PAGE_FLIP=1



5 Secure ROM services

The secure boot is the code that is always run first when MAX32550 starts up. This initiates the chain of trust that applications execution relies on: the secure boot guarantees to the applications the platform they are running on is trustworthy and is the one expected to be; the secure boot also guarantees that the applications running on the platform are only the authorized ones, so only trusted applications can run on the platform.

Objective of the secure boot is to provide high security strength to the platforms while keeping flexibility and ability to adapt to various customer needs. In that perspective, the secure boot has multiple mechanisms.

The first mechanism is the verification before launch of user applications: this verification guarantees that only authenticated applications can run on the platform (see section 5.3).

The second mechanism is the secure programming of the platform memories (OTP, flash), including applications secure updates. The loading of applets in the internal RAM memory allows running tests programs (see section 5.2).

The security the secure boot implements to provide trust is based on standard public-key cryptography. The MAX32550 secure boot uses customer public key to verify the applications and to authenticate the updates (see section 5.1).

Beyond these security mechanisms, the MAX32550 provides means for erasing the battery-backed 256-bit AES key that secures the NVSRAM. This “key wipe” mechanism ensures that the AES key is not exposed in case of attacks attempts.

5.1 Customer key

The MAX32550 secure boot performs authentications using ECDSA P-256 (see[7]), with 256-bit ECC key pairs and SHA-256 secure hash function (see [7]). These algorithms require keys values available on the platform.

A small PKI is defined for MAX32550 secure boot security. This PKI is based on a Maxim Integrated Root Authority key, whose public key is stored in the ROM memory. Therefore, any customer key shall be certified by this Root key before use on the platform.

The chips are delivered to customers without any customer key. The first, mandatory step for the customer is to program its certified 256-bit ECC public key in the internal OTP memory (nothing else can be done before this step).

The sequence for programming the customer key is the following:

1. The customer securely generates its ECC key pair CRK, the public part is the customer key to be programmed in MAX32550 OTP.
2. The customer exports the CRK public key and sends it to Maxim Integrated, for certification.
3. Maxim Integrated generates a certificate for this CRK public key, using the Maxim Integrated Root Key, the MRK. The CRK certificate contains the CRK public key and a digital signature of this key. Maxim Integrated sends the CRK certificate back to the customer,
4. The customer uploads the CRK certificate to the MAX32550 platform, using the secure boot protocol (Secure Communication Protocol, SCP),
5. The customer resets the chip

At this stage, the platform secure boot is now able to perform secure updates (see section 5.2) and to launch applications from internal memory (see section 5.3), using the customer key, the CRK.

Customer attention is drawn on the fact that the secret part of its ECC key pair CRK shall be handled (stored and used) securely. Maxim Integrated cannot guarantee the chain of trust if this key is compromised, therefore Maxim Integrated recommends the usage of a Host Security Module (HSM) or an equivalent hardware device providing physical protection to the secret part of the ECC key pair and complying with PCI PTS keys protection requirements.

Note: Maxim Integrated protects its own MRK secret part by highly secure means, including a high-security grade HSM, compliant with FIPS 140 and PCI PTS security requirements.

5.1.1 CRK certification

- CRK generation: The customer generates his own CRK keypair, using PCI PTS compliant tools and procedure: this key is a 256-bit ECC keypair, using the P-256 curve
- PGP key ceremony:
  - A customer identified person (e.g. the project manager, the security manager, …) sends by email his PGP key to his Maxim Integrated Business Manager and confirms it through another channel (skype, phone call)
  - The Maxim Integrated Business Manager sends by email the `crk.certificate` PGP public key
- CRK sending: The customer sends the CRK public key value to Maxim Integrated by email:
  - The CRK is inserted in the body email
  - The CRK format is:
    - Hexadecimal format: 64 digits using `0123456789ABCDEF` digits,
    - Respect of the notations described in sections 1.1 and 1.2, i.e. natural format; the number 8903214_{10} is coded as $87DA2E_{16}$
  - The email address where to send the public key is `crk.certificate@maximintegrated.com`
  - The email shall explicitly contain the customer name
  - The email shall be signed by the sender, using PGP
- CRK certificate reception
  - The sender receives an email signed by the `crk.certificate` PGP key
  - This email has an attached zip file containing the SCP packets (a set of .packets binary files and a .list file)
  - These packets are ready for use with the `serial-sender` tool. These packets contain the full session to be sent to chips in #3.These packets write the CRK and move the chips to #4.

5.2 Secure loader

A Host, typically a PC, is able to communicate with the MAX32550 secure boot using a secure protocol on either the serial link or the USB link. This protocol, the SCP, enables the Host to send commands to the secure boot. Every command is authenticated by the secure boot before being interpreted and executed. Main commands are those related to memories programming.

5.2.1 SCP Session

The SCP is a session-based protocol (see Figure 3). The Host requests session opening to the MAX32550 secure boot and then sends the commands to it. Each command shall be signed using the CRK private key before sending. The digital signature is verified by the MAX32550 secure boot before its interpretation and execution.

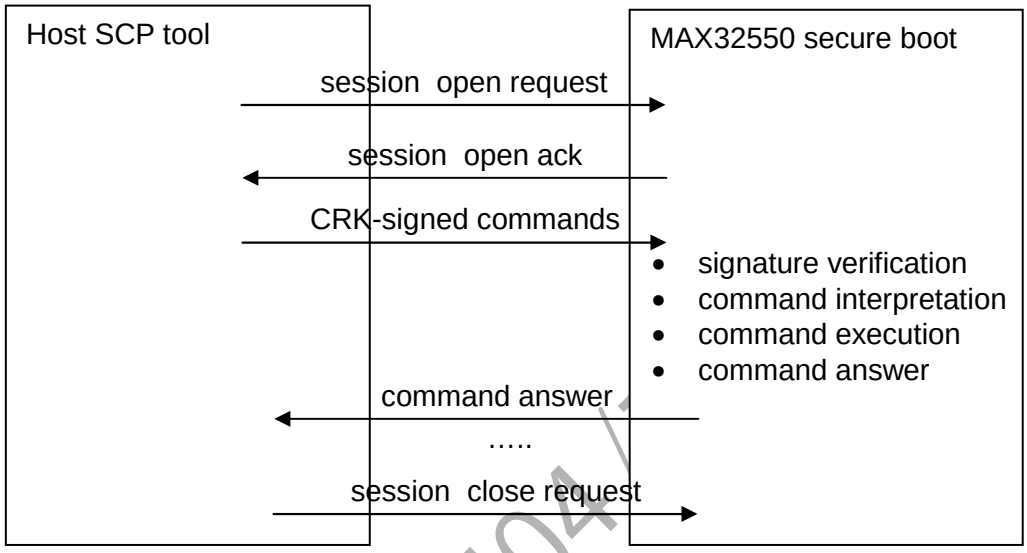


Figure 3 SCP session

5.2.2 Packets signature

The commands packets signatures may be computed on-the-fly or before the packets are sent. It enables to split the step for secure handling of the CRK and the step for effective download operation. It relaxes therefore some constraints on the security of the premises.

5.2.3 OTP programming

The MAX32550 OTP internal memory is mainly used for platform configuration. As this depends on the customer platform characteristics, it is up to the customer to configure this memory. The SCP provides the ability to program the OTP memory.

Programming the CRK certificate in OTP is achieved through a dedicated command.

5.2.4 Applets programming

The SCP provides a framework to run applets. An applet is a small program loaded into the internal RAM and that can implement several kinds of needs (tests, keys generation, drivers).

Once this specific small program written, it is loaded in MAX32550 internal RAM by the SCP and ready to be run.
This framework is described in Figure 4:

- the path ‘1’ represents the loading in MAX32550 internal RAM of the applet, using SCP command in a SCP session,
- once the applet is loaded, the applet can be run from invocation by SCP commands,
- the path ‘2’ represents the data the applet can handle through SCP commands.

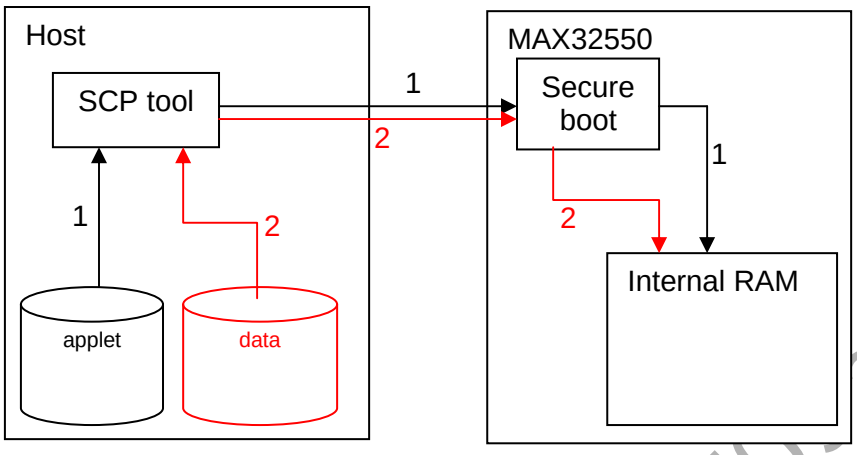


Figure 4 application programming

Note1: a framework for applets development is provided to customers willing to develop or customize their own applets.
Note2: the applet mechanism is also optionally available during the load-and-launch process, the applet being stored in the NVSRAM.

5.3 Secure boot

The MAX32550 secure boot verifies any application before launching it (see Figure 5). The verification is performed on the application digital signature computed with the CRK private key.
The CRK digital signature is stored within the application in the internal non-volatile memory (e.g. previously programmed by the secure loader mechanism).

The MAX32550 secure boot first retrieves (step '1') configuration parameters stored within the internal OTP memory (previously programmed by the secure loader mechanism). Using these parameters, the MAX32550 secure boot accesses (step '2') the application and its signature. These parameters are called the "source" parameters as they help the secure boot to access to the application origin, i.e. the non volatile storage memory.

Once the verification is performed using the CRK public key (step '3'), the secure boot runs the application from the flash (XIP). Parameters for this execution are available in the application header.

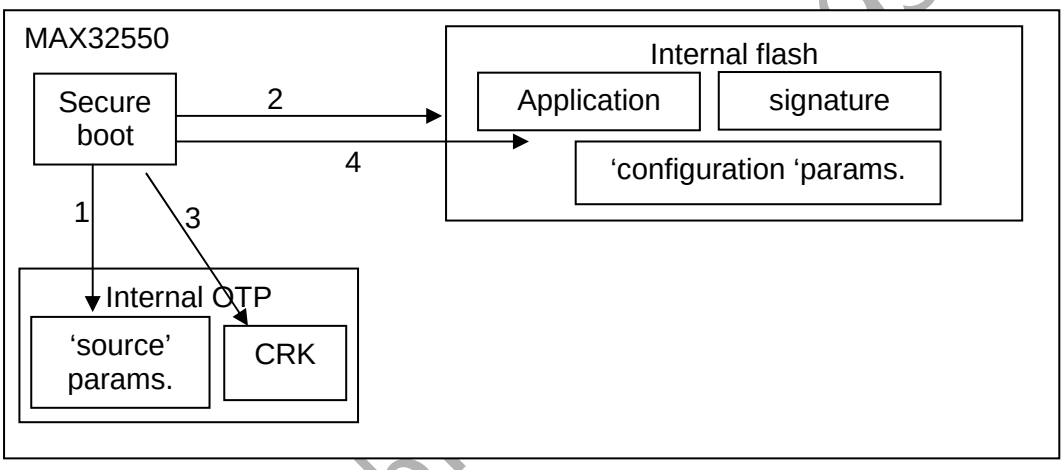


Figure 5 MAX32550 secure application execution

5.4 Key wipe mechanism

The MAX32550 ROM code is able to erase the battery-backed 256-bit AES key upon certain conditions. This "key wipe" mechanism ensures that the AES key is not exposed in case of attacks attempts.

5.4.1 A and B1 revisions

On MAX32500 revision B1, A3 and before, this "key wipe" mechanism is always enabled. The erasure is then triggered when:

- The ROM checksum verification fails
- The ROM cryptographic self-tests fail
- The ROM life cycle retrieval fails
- The USN checksum verification fails
- The application secure header or signature verification fails.

5.4.2 B2 and later revisions

On MAX32550 revision B2 and after, the "key wipe" mechanism is optional and can be activated by a OTP configuration (see Table 1).

Furthermore, the “key wipe”, if activated, can be triggered after a certain number of errors.
This mechanism also requires the use of the two last bytes in the NVSRAM.
The process is the following:

- If the OTP value is not configured, the “key wipe” mechanism is not activated
- Else
    - This OTP value is considered as the number of detected errors before performing the “key wipe”. The 0 value means the “key wipe” is performed once an error is detected
    - If there is one firmware image in the flash that has a correct signature
        - The NVSRAM counter is set to 0
        - The ROM code jumps to the firmware image
    - Else if none of the digital signatures of the firmware images in the flash is correct
        - Then the NVSRAM counter is incremented
        - If the NVSRAM counter is great or equal to the OTP value
            - The “key wipe” mechanism is performed
    - Else reset

- Notes:
    - “none of the digital signatures of the firmware images is correct” means
        - one image at least need to be present
        - if no image (i.e. no security header) is present, there is no “key wipe”, only a reset.
    - it is up to the customer to program this OTP value; the OTP value shall be fed with its redundancy
    - if the OTP value is not consistent, i.e. the redundancy is not correct, there is no “key wipe”
    - even when a “key wipe” has occurred, the update mechanism is still possible meaning the platform can be reused and reactivated (firmware loading, AES key generation).
    - The NVSRAM two-byte location includes some redundancy. If this redundancy check fails, there is no “key wipe”.
    - The OTP value includes some redundancy (OTP check value and complement to 1). If this redundancy checks fail, there is no “key wipe”.

6 Quick start

This chapter describes the steps to perform for running an example application on a MAX32550 SoC. It especially describes how to accomplish the different steps related to life-cycle and security management. This is typically the steps that the Secure ROM package helps for (see [3]).

6.1 Configuring MAX32550

MAX32550 SoCs are delivered on purpose in a locked state and no application can be run under these conditions. It is up to the customer to unlock the SoCs and make them usable.
There are three major steps to achieve that:

- Program the customer ECC-256 public key: this public key will be used to check that an authorized, genuine application has been loaded before its execution,
- Set up the user OTP area for secure update link choice,
- Program the signed application in the non volatile internal memory.

Obviously, each of these three steps includes several smaller steps. They will be described in the following sections.

Note: the wording “application” covers any kind of firmware binary that is run immediately after the MAX32550 ROM code. This application may be a 2nd level loader, a demo standalone application or even a full OS. It mainly depends on customer needs and memories configuration.

6.1.1 Program the customer ECC-256 public key

The customer has to program his ECC-256 public key on the MAX32550 SoC. This key will be written in the customer OTP area and used for application authentication. It has to be signed by Maxim before being programmed in the customer OTP.

To achieve this objective, the following steps shall be performed:

1. The customer shall generate a ECC-256 key pair,
    - This key is named the CRK, Customer Root Key; here is an example (32-byte value for the secret,32-byte for the x affine coordinate, 32-byte value for the y affine coordinate):

      ```
      7ac88a77095ce13e593b83904064f98351df9ed430eb143c4abc55a984e57f39
      a823c8857948dc688f3a3ef3f6f220a514f05c2c6c1cef8c9f2f8df11dcf0142
      3be124619cbbeb51e985328e8e33d321cade19628cc0db43304a7b27f2db8efe
      ```

    - The key generation and handling shall require the maximum level of security (e.g. generated within a HSM), in order to comply with security requirements (e.g. PCI PTS, FIPS 140-2, …),
    - During development, it is recommended to customers to use a test key. Either the one provided by Maxim Integrated in the Secure ROM package or a key generated by any means: this key does not need to be securely generated and used.
2. The customer shall send its ECC-256 public part, i.e. the x and y affine coordinates to Maxim Integrated (this step is not required if the customer uses the test key provided in the Secure ROM package, as it is provided already signed):
    - The format is ideally a text file, made of two lines, containing a32-byte hexadecimal value of the x affine coordinate and a 32-byte value of the y affine coordinate; here is an example:


```
a823c8857948dc688f3a3ef3f6f220a514f05c2c6c1cef8c9f2f8df11dcf01
42
3be124619cbbeb51e985328e8e33d321cade19628cc0db43304a7b27f2db8e
fe
```

3. Maxim Integrated signs this public key and sends back the corresponding digital signature, using a text file format (i.e.64-byte hexadecimal value of the digital signature); Maxim Integrated also sends SCP packets (i.e. a set of files) to be downloaded to the MAX32550 SoC,
4. The customer uses the tool **serial-sender** (see 7.3.3) for SCP packets download.

At this stage, the MAX32550 SoC contains the customer CRK. The device will now execute code if and only if it has been signed by the customer private key i.e. the customer fully controls any download performed on the MAX32550 SoC. Next step is the customer OTP configuration.

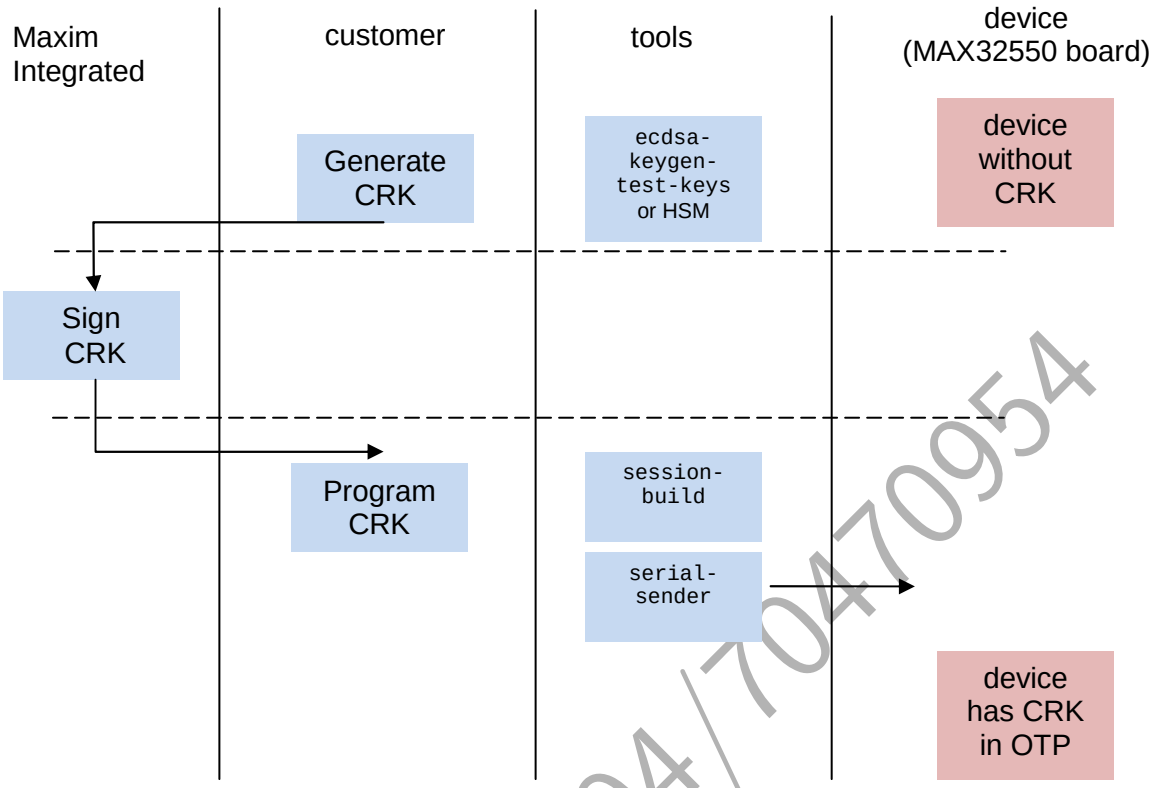


Figure 6 CRK generation and programming

6.1.2 Updating CRK

Latest MAX32550 versions (from A3) allow updating CRK already stored. This feature does not erase and replace, nor overwrite existing CRK. Process is to program a new CRK through SCP command (REWRITE-CRK) elsewhere into User's OTP.

- Note that, as for 1st CRK programming, if area is not "*virgin*", programming is going to fail.
- For MAX32550-A3, REWRITE-CRK SCP command has a similar structure than current WRITE-CRK SCP command; new CRK and its signature are sent to ROM Code.
- For MAX32550-B1 and further versions, REWRITE-CRK SCP command has an additional field. This field is for "CRK to be replaced" (without it signature). ROM Code checks then that CRK stored into current area matches "CRK to be replaced"; this in order to avoid a third party to re-program easily the platform with its own CRK.
    - Before signing the new CRK, Maxim Integrated validates that the entity asking for the signature is the owner of the previous CRK, to be replaced.

6.1.2.1 Choosing CRK to use

During SCP communication or SLA digital signature verification, CRK is (verified and) used; but both features do not know about CRK1/2, they only need **CRK**.
Therefore ROM Code always checks:

- CRK2 is (well) programmed into OTP, then **CRK2** is CRK.
- CRK2 is not programmed into OTP (or corrupted), then **CRK1** is CRK.

6.1.3 Set up the user OTP

The MAX32550 user OTP is not programmed by default.
This OTP configuration is performed using the SCP protocol. As using the CRK private part for SCP packets signatures computation, the key handling shall require the maximum level of security (e.g. generated within a HSM), in order to comply with security requirements (e.g. PCI PTS, FIPS 140-2,…). In test conditions, the **session-build** tool can be used for SCP packets computation, by using ECDSA CRK test key.

6.1.3.1 User OTP mapping

In MAX32550 OTP, data granularity is one line, i.e. **8 Bytes**. Please refer to [1] and **Error! Reference source not found.** for OTP characteristics.

offset	bit_{63}	bit_{62} .. bit_{15}	bit_{14}..bit_0	
`userbase+`	Lock bit	data	CV	comments
`0000`	1b	CRK_1–512bits ECDSA $key_{(LSB)}$	xxxx	
…	…	…	…	
`0050`	1b	'0' from bit_{62} to bit_{47} and **Last CRK_1's 32bits$_{(MSB)}$** from bit_{46} to bit_{15}	xxxx	
`0058`	1b	**CRK_1 signature** – 512$bits_{(LSB)}$	xxxx	
…	…	…	…	
`00a8`	1b	'0' from bit_{62} to bit_{47} and **Last CRK_1 signature's 32bits$_{(MSB)}$** from bit_{46} to bit_{15}	xxxx	
`00b0/HL`	1b	'0' from bit_{62} to _bit_{47} and **HHA location 32bits** from bit_{46} to bit_{15}	xxxx	HHA or Magic Value. If Magic Value, platform goes into shutdown mode
`00b8`	1b	bit_{62} to bit_{55} **Boot Source ID**, '0' from bit_{54} to bit_{47} and bit_{46} to bit_{15} **Binary location 32bits**	xxxx	**Byte 5** is **Boot Source ID** indicating if storage memory to use is the one from **Internal Flash**$_{(0)}$ or **SPI**$_{(1)}$ – **Byte 3 - Byte 0** is **Binary Location** address
`00c0`	1b	'0' from bit_{62} to bit_{47} and **UART0 configuration 32bits** – bit_{46} to bit_{15}	xxxx	
`00c8`	1b	**CRK_2** – 512bits ECDSA $key_{(LSB)}$	xxxx	
…	…	…	…	
`0118`	1b	'0' from bit_{62} to bit_{47} and **Last CRK_2**'s 32$bits_{(MSB)}$ from bit_{46} to bit_{15}	xxxx	
`0120`	1b	**CRK_2 signature** – 512$bits_{(LSB)}$	xxxx	
`0170`	1b	'0' from bit_{62} to bit_{47} and **Last CRK_2 signature**'s 32$bits_{(MSB)}$ from bit_{46} to bit_{15}	xxxx	
`0178`		0 from bit_{62} to bit_{31} and **Key Wipe Counter Limit 16bits** – bit_{30} to bit_{15}		Key Wipe Counter Limit is coded on 16 bits with LSB for actual limit value, and MSB for its *complement*. For instance, if limit is `0x04`, value stored in OTP should be `0xfb04`.
`0180`		*free to use*	Xxxx	

Table 1 User area OTP mapping

Note 1: For more details about the session-build tool use (and especially the `write-otp` command), see 7.3.2 and [2].

6.1.4 Program the application in embedded flash memory

The application to be run on the MAX32550 SoC has to be stored in the embedded flash. The handled application binary file is not the file result of the build process, as a specific format is used (let's call it MAX32550 application format); the process for application format conversion is described below. For a correct execution, some prerequisites are mandatory:

- The user OTP shall be correctly configured (see section above),
- The application has been developed using the 2nd level application (SLA) framework. The framework package includes the documentation, an example of application and the tools for building (Eclipse, gcc tool chain),
- The application has a specific format:
    - It shall be signed with the CRK,
    - The defined application header shall be correctly configured,
        - The header configuration contains information about the binary start address, application version, … (more details in [1])
    - For these two operations, an access to the CRK private part is needed; the key handling shall require the maximum level of security (e.g. generated within a HSM), in order to comply with security requirements (e.g. PCI PTS, FIPS 140-2). In test conditions, the **ca-sign-build tool** can be used for application signature and header configuration (see 7.3.1). The `load_address` and the `jump_address` can be found in the `.map` file generated at build by the SLA framework or by user's own development environment.
      Note: the COBRA development framework (see [4]) includes these operations for the test development CRK.

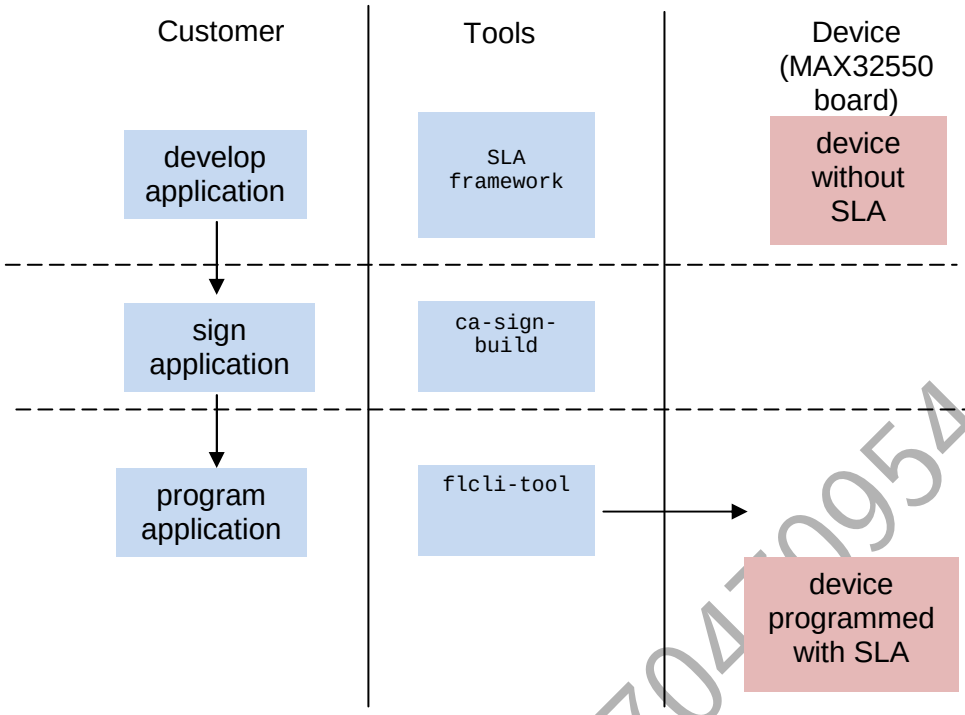
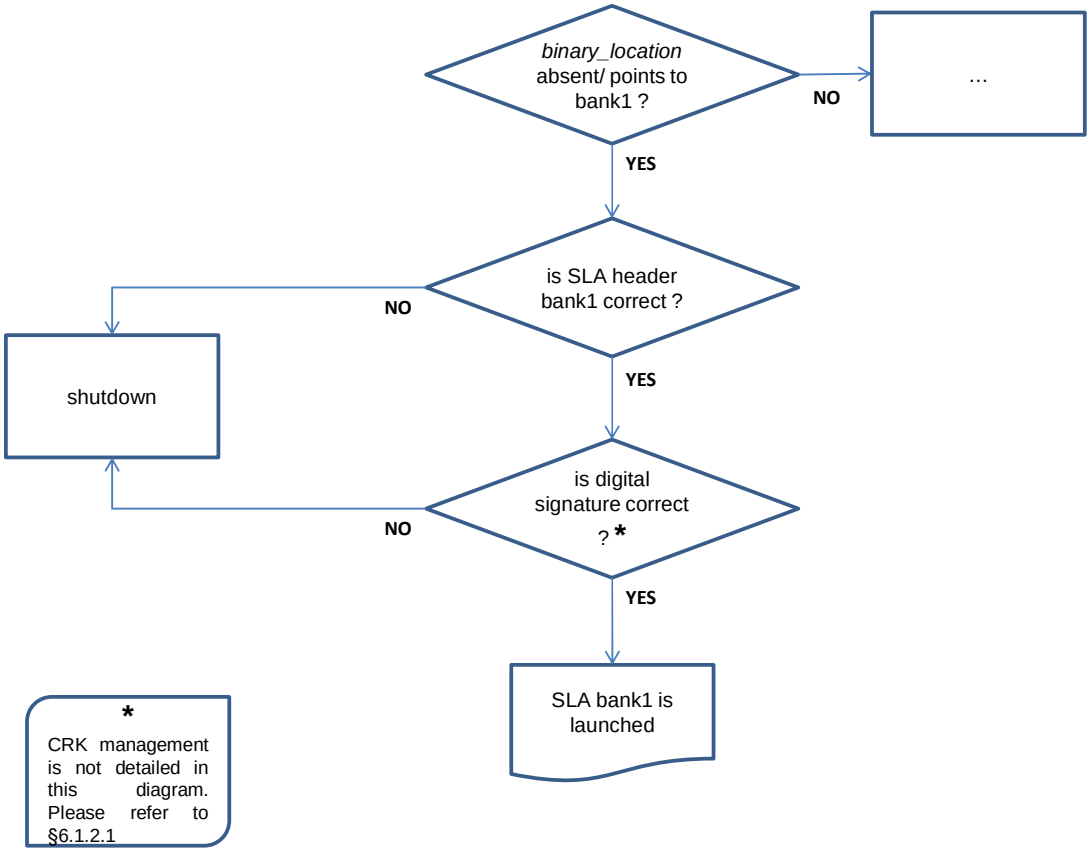


Figure 7 SLA development and programming

6.1.5 Load & Launch sequence

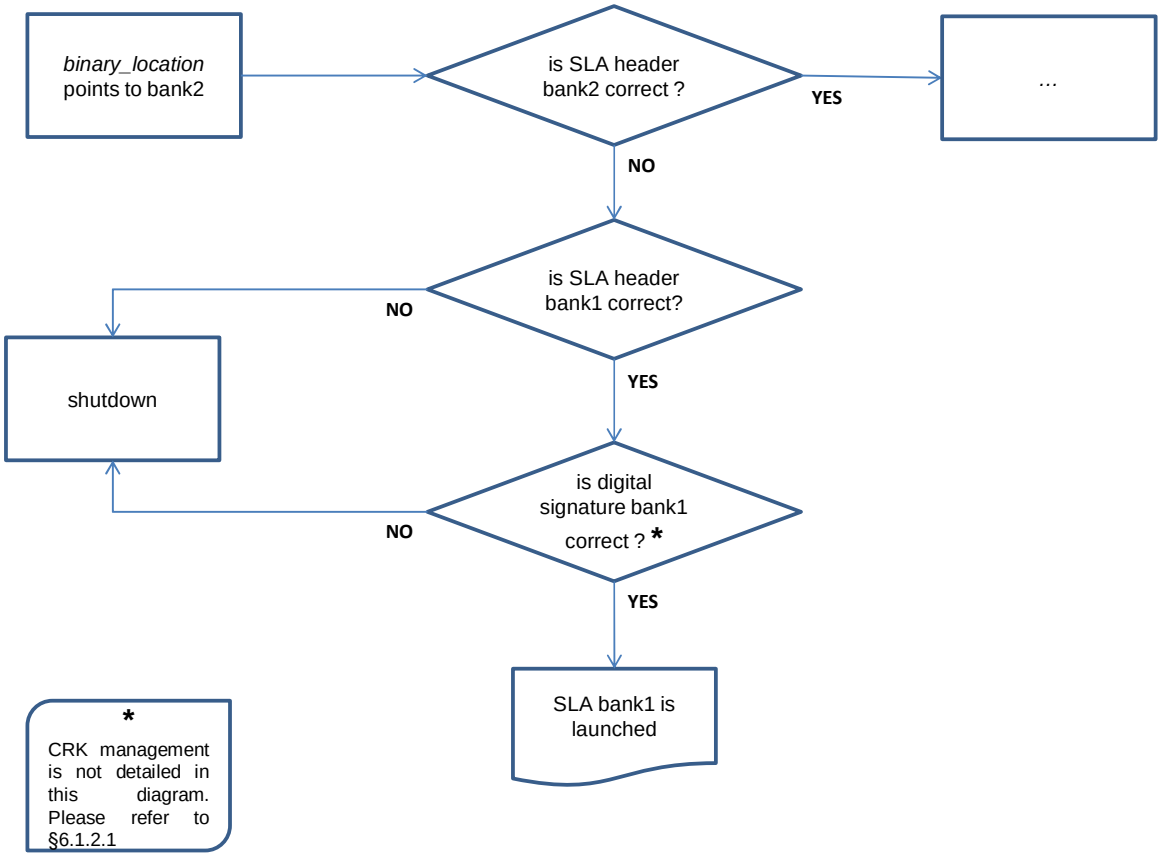
Remember that internal flash is made of 2 banks (please refer to §4.4). Behavior is conditioned by "*binary_location*" parameter from User's OTP (please refer to §6.1.3.1).

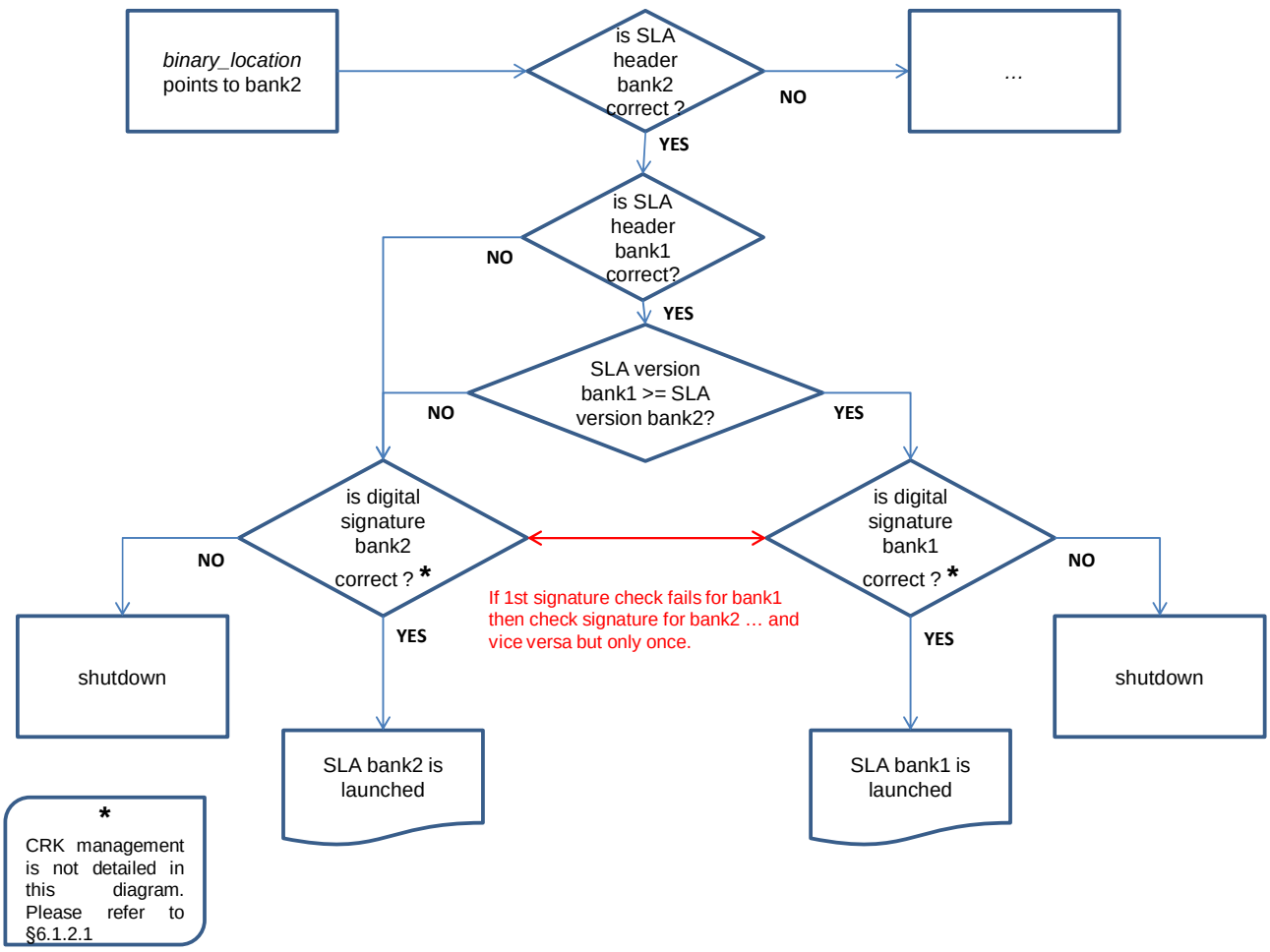
binary_location is absent or points into 1st internal flash bank.



Note that in this case, internal flash is seen as one block (bank1 + bank2).

binary_location points into 2nd internal flash bank:





6.1.6 Conclusion

If every step described above has been correctly performed, at next reset, the MAX32550 SoC shall run the application.

7 Secure ROM tools

The tools are used for guaranteeing security for ROM code main mechanisms:

- download,
- keys management,
- application execution.

Their purposes are then to allow the user to:

- customize the embedded platform relatively to the user expectations (secure memory content, UART configuration), keys use and policy (secret keys, public keys),
- develop without constraints in a secure context,
- learn about keys management and functioning.

Note1: It is important to notice that tools using secret/private keys are not compliant solutions with PCI-PTS certification and are only to be used in development/testing context.

7.1 Supported platforms

These tools are running on

- Cygwin v1.7.7 64-bit on Windows 7 64-bit
- Linux Ubuntu v12.04 LTS

7.2 Data files format

ROM code host tools handle data (keys, configuration parameters) through use of files (mainly text files). In this section, main data files formats are provided.

7.2.1 ECDSA private key file

The type is `ecdsa-privkey-file`.
This text file contains three lines:

- 32-byte hexadecimal-coded secret
- 32-byte hexadecimal-coded x affine coordinate
- 32-byte hexadecimal-coded y affine coordinate

Example:

```
7ac88a77095ce13e593b83904064f98351df9ed430eb143c4abc55a984e57f39
a823c8857948dc688f3a3ef3f6f220a514f05c2c6c1cef8c9f2f8df11dcf0142
3be124619cbbeb51e985328e8e33d321cade19628cc0db43304a7b27f2db8efe
```

The corresponding decimal value for the secret above is:

```
5553649259032970591845465084231497643775489832492785515079727354204812827013
7
```


7.2.2 ECDSA public key file

The type is `ecdsa-pubkey-file`.
This text file contains two lines:

- 32-byte hexadecimal-coded x affine coordinate
- 32-byte hexadecimal-coded y affine coordinate

Example:

```
a823c8857948dc688f3a3ef3f6f220a514f05c2c6c1cef8c9f2f8df11dcf0142
3be124619cbbeb51e985328e8e33d321cade19628cc0db43304a7b27f2db8efe
```

7.2.3 ECDSA signature file

The type is `ecdsa-signature-file`.
This text file contains one line:

- 64-byte hexadecimal-coded signature, computed using ecdsa-p256-sha256

Example:

```
5b5c553b6405a3bb432d76566e9e480798b27dcda0d69af22cc230df89cf5483
ab99f0a36e0f2024f151243a26186fab13e7e8067d001c98c13d4ebcc3d86bea
```

7.2.4 ECDSA signed public key

This file format is the combination of the ECDSA public key file format and ECDSA signature file format.
The type is `ecdsa-sign-pubkey-file`.
This text file contains three lines:

- 32-byte hexadecimal-coded x affine coordinate
- 32-byte hexadecimal-coded y affine coordinate
- 64-byte ECDSA signature

Example:

```
a823c8857948dc688f3a3ef3f6f220a514f05c2c6c1cef8c9f2f8df11dcf0142
3be124619cbbeb51e985328e8e33d321cade19628cc0db43304a7b27f2db8efe
5b5c553b6405a3bb432d76566e9e480798b27dcda0d69af22cc230df89cf5483
ab99f0a36e0f2024f151243a26186fab13e7e8067d001c98c13d4ebcc3d86bea
```

7.2.5 binary file

The type is `binary-file`. This binary file is made raw bytes.

7.2.6 s19 file

The type is `s19-file`.
This text file contains one or multiple lines:

- only the S3-flagged lines are considered,
- the created s19 files have correct CRC byte at the end of each line,

7.3 Tools

The tools are used in a logical sequence.
Except the ones not using any option, every tool has a `.ini` file which contains the list of the options with already determined values.

7.3.1 Customer application signature

The customer application needs to be signed before being programmed on the chip, as this signature is controlled before execution. This tool adds the SLA header (as specified in ROM code specifications) and signs the resulting file. The result file is compliant with ROM code “Load and Launch” routines.

NAME

ca_sign_build: build binary file including ECDSA signature and SLA header from customer application binary file.

SYNOPSYS

ca_sign_build [ecdsa=<ecdsa-privkey-file name>] [algo=ecdsa] [ca=<application binary-file>] [sca=<signed application binary-file>] [jump_address=<4-byte hexadecimal value> [arguments=<string>]

OPTIONS

these options are those described in the ROM code 2nd level application header format:

- ecdsa=<ecdsa-privkey-file name>:256-bit ECC key used for application signature
- algo=ecdsa: this field describes which algorithm has to be used (the tool supports also rsa for MAX32590 use),
- ca=<application binary file>: the origin (input) binary file
- sca=<signed application binary file>: the final (output) signed+header binary file; it contains:
  - the header,
  - the input binary file and
  - the appended signature
- jump_address=<4-byte hexadecimal value>: : the jump address of the binary application,
- arguments=<string>: the arguments to be provided to the final application through argc,argv; the string is limited to 10KB by the tool,
- version=<4-byte hexadecimal value>: identifies the ROM code version, i.e. the header version ROM code is able to support,
- verbose=<yes|no>: this option states if verbose mode is activated during execution, or not.

LIMITATIONS

- the arguments string length is limited to 10KB

EXIT CODES

- 0: everything is OK
- 1: failure

EXAMPLES

- **ini file:**

```
# lines starting by a # are considered as comments
#the 256-bit ECC key used for application signature
ecdsa=casignk.key
#the origin (input) binary file
ca=appli.bin
#the chosen algorithm
algo=ecdsa
#the final (output) signed+header binary file
#it contains the header, the input binary file and the appended signature
sca=appli.sbin
#the jump address of the binary application
jump_address=01020304
#the binary length is automatically computed
#arguments
```



```
      argv="string of chars"
      #arguments length is automatically computed
      #verbosity level
      verbose=yes
$./ca_sign_build.exe ecdsa=casignk.key algo=ecdsa ca=appli.bin sca=appli.sbin
load_address=01020304 jump_address=02030405 verbose=yes
```

7.3.2 SCP session build

The SCP session tool computes offline the SCP frames corresponding to the provided parameters. This tool is used for several platforms and some options only apply on some of them (e.g. ecdsa_file applies does not apply for MAX32590, flash_size_mb does not apply for MAX32550).

NAME

session_build: builds offline set of files containing the packets sent during a SCP session between the host and ROM code.

SYNOPSYS

session_build [session_mode=<mode>] [algo=ecdsa] [verbose=yes|no] [output_file=<file-name-radix] [pp=<protection-profile>] [ecdsa_file=ecdsa-privkey-file] [script_file=<file name>] [addr_offset=<address>] [chunk_size=<byte size>]

OPTIONS

- session_mode=<mode>: this option states the SCP session mode, i.e. SCP_ANGELA_ECDSA,
- verbose=<yes|no>: this option states if verbose mode is activated during execution, or not,
- output_file=<file-name-radix>: this option states the radix for the generated files (packets and logfile),
- pp=<protection-profile>: this option states the protection profile, which is ECDSA,
- ecdsa_file=<ecdsa-privkey-file >: this option states the ECC private key file name,
- script_file=<script file name>: this option states the script file name to be used for the generated SCP session,
  - scripts syntax is the following:
    - lines starting by a '#' are considered as comments and are not processed,
    - each line contains only one SCP command,
    - some commands require one parameter (e.g. value, file name)
    - the list of available commands is the following:
      - write-file <s19-file>
        this command implicitly performs an accurate erase before programming,
        this command implicitly performs a verify after programming,
      - write-only <s19-file>: this command performs only the data write operation (no erase before, no verify after)
      - erase-data <start-addr> <length>: this command performs an erase on the memory area defined by the 'start-addr' address and the 'length' number of bytes
      - write-otp <hex offset:2bytes> <hex data value>
      - write-crk <ecdsa-signed-pubkey-file>
      - write-timeout <target char> <hex timeout value:ms>: this command sets the timeout value up in milliseconds, for the targeted interface, either UART0 (char '0') or USB ('U') or VBus Detect (‘V’)

        - kill-chip
        - execute-code <hex address:4bytes>
- addr_offset=<address>: this option states the initial offset for s19 file download, in internal memory.
- chunk_size=<byte size>: this option sets up the size of the payload buffer used for write-data packets; for alignment constraints and best performances, this size shall be 4094 for A3 but 15354 for B1.

OUTPUTS

- the output_file radix value is used to generate:
    - a text log file containing the details of each packet, named output_file_radix**.log**,
    - a binary file containing the bytes for each packet:
        - A packet issued by the host is tagged .host., named output_file_radix.host.<file-number>.<action>.packet
        - A packet issued by the bootloader is tagged .bl., , named output_file_radix.bl.<file-number>.<action>.packet,
        - where <file-number> is an integer value incremented by one for each new binary file created and <action> corresponds to the kind of operations represented by the binary frame (e.g. connection request, hello reply, write mem, …).

LIMITATIONS

- every SCP command is not implemented in the tool, please refer to the script available commands list,
- the addresses used in the s19 file shall be based on the 0x00000000 value.
- The chunk_size value shall be 4094/15354.
- The script-file shall end with a end-of-line, i.e. the last command shall be entered with a CR

EXIT CODES

- 0: everything is OK,
- 1: failure.

NOTES

- this tool can be used for other parts (e.g. MAX325xx, MAXQ1852), with other configuration parameters; only those relevant for the MAX32550 are described here

EXAMPLES

```
$cat script.txt
write-file myapplication.s19
write-otp 000A 040506070809
write-bpk 0004 0102030405060708090A0B0C0D0E0F00
write-crk crk_test.signpub
write-timeout 0 012C
execute-code 01020304

$session_build session_mode=SCP_ANGELA_ECDSA ecdsa_file=crk.key  verbose=yes
output_file=session script=script.txt addr_offset=00000000 chunk_size=4094
```

the log file `session.log` and packets files `session.host.1.connection_request.packet, session.bl.2.connection_reply.packet, session.host.3.ack.packet, session.host.4.hello_request.packet, …,` are created.

7.3.3 Packets serial sender

The `serial_sender` tool uses packets generated by the `session_build` tool in order to send the `.host.` ones to the bootloader on the serial port and to wait for the `.bl.` ones.

NAME
serial_sender sends signed packets to the bootloader on the serial link. This tool can be used as test tool for bootloader and validation tool. FILENAME contains both sent and received packets. Those last ones are used for verification.
SYNOPSYS
Usage: serial_sender [options] FILENAME
OPTIONS
--version show program's version number and exit
-h, --help show this help message and exit
-s SERIAL, --serial=SERIAL
define the serial port to use
-v, --verbose enable verbose mode
-l, --license display license
--list-serial display available serial ports
-b, --bl-emulation emulate the bootloader
-t SECONDS define the timeout used for the commands answers from the device, expressed in seconds. Default value is 35s. This value shall be adapted in function of the flash area to be erased, as this erasure may take some time (so, the timeout shall not be too short).
LIMITATIONS

- the FILENAME contains the list of packets files to be processed. The files are processed by the tool following the sequential order. So, the files shall be correctly sorted at file list building (see the example below).
- The tool sends packets only once, so the connection request is sent only once; therefore, the platform shall be reset before starting the tool (and correctly synchronized).

ERRORS

- the serial_sender checks the packets received from the bootloader and raises an error if the data are different than the ones expected, except for the HELLO_REPLY packet (as it may vary on the USN).

EXAMPLE

```
$ls session.*packet|sort -t '.' -n +1 >session.list
or
$ls session.*packet|sort -t '.' -n -k1 >session.list
$serial_sender --list-serial
COM1
COM3
COM8
$serial_sender -s COM1 session.list -v
```


7.4 Examples of use

Here is a simple scenario of when tools are used is presented.

- **Keys handling**
- generation of ECDSA keys (generally done once)
- **platform use**
- customer application signature: ca_sign_build
- customer crk signature
- **platform programming**
- signed customer application programming: session_build+serial_sender

8 PCI PTS compliance

- Cryptographic algorithms compliant with B11
- Secure boot (integrity check) is compliant with B1
- Secure boot (update) is compliant with B4
- Keys management compliant with B11

9 Annex A: cryptographic support

9.1 ECDSA

This algorithm is used for digital signatures. The public part is used for digital signature verification while the private part is used to sign data.
The digital signature algorithm is ECDSA, with curve P-256 (see [7]).
It is used in SCP public session security, with 256-bit keys, for certificates authentication and for application signature verification before execution.

9.1.1 SHA-256

This algorithm is used for digital signatures with ECDSA.

10 Annex B: ECDSA signature

ECDSA Keys are 256-bit long. The principle of one key for one use is enforced.
The ECDSA keys are used for digital signatures.
A digital signature scheme typically consists of three algorithms (extracted from [2]):

- *A key generation algorithm that selects a private key randomly from a set of possible private keys. The algorithm outputs the private key and a corresponding public key.*
- *A signing algorithm which, given a message and a private key, produces a signature.*
- *A signature verifying algorithm which given a message, public key and a signature, either accepts or rejects the message's claim to authenticity.*

The diagram below (Illustration 3 extracted from [2], this file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license, the author is acdx, see [2] for more information) describes a digital signature scheme, applicable to the ECDSA signatures.

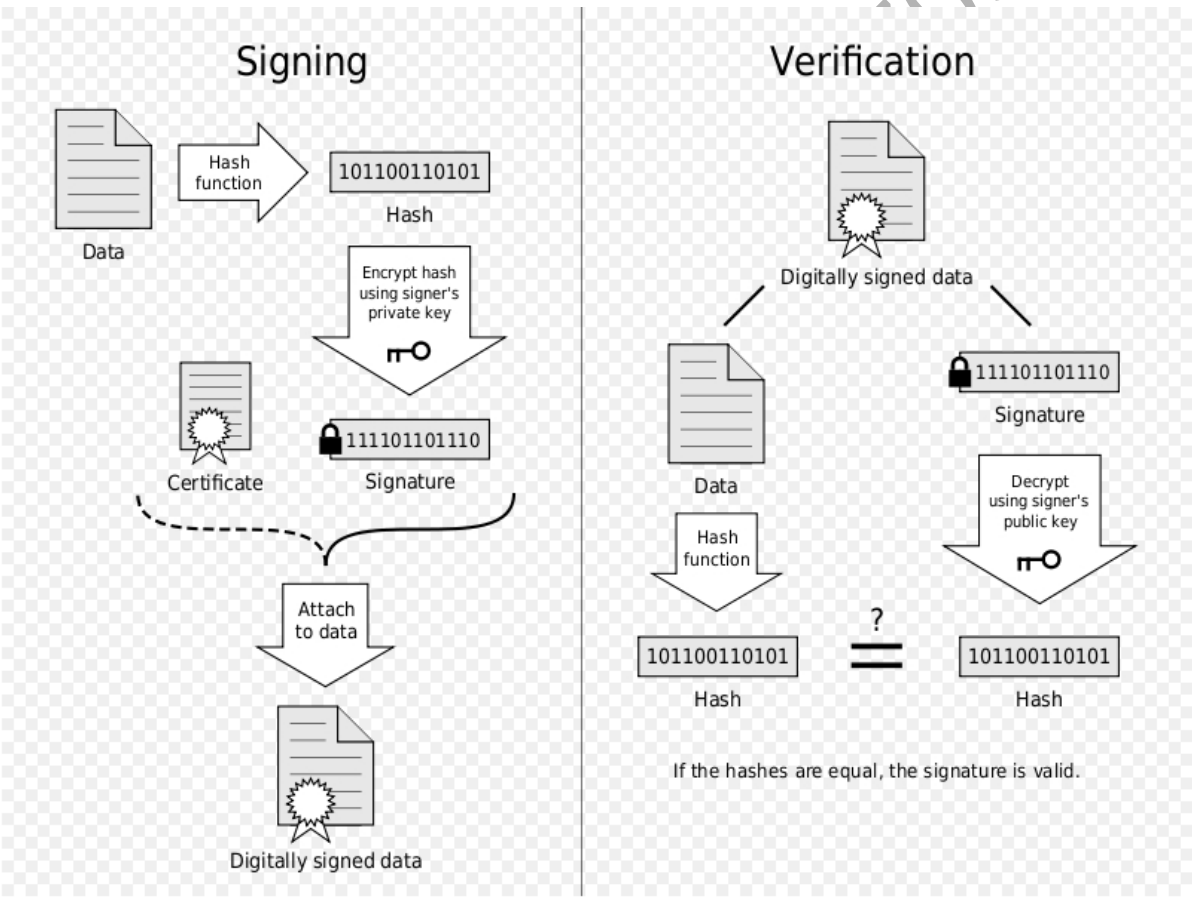


Figure 8 Digital signature scheme

Maxim Integrated

http://www.maximintegrated.com support:
goto www.maximintegrated.com/support

