Deeptrust for Cortex-M

SPEC98T17
Revision E
06/19/2017

Maxim Integrated, Inc.
160 Rio Robles
San Jose, CA 95134

Disclaimer

LIFE SUPPORT POLICY

MAXIM'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF MAXIM INTEGRATED PRODUCTS, INC.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

Contents

1. Document details

1.1 Release Notes

Revision	Date	Description
A	04/12/2016	Initial Release
B	03/01/2017	Add Debug Box Refine interrupt handling discussion Introduce event management Refine key management
C	02/02/2017	Clarifications in the security architecture description
D	19/05/2017	Reflect latest modifications
E	19/06/2017	Improvements, see note 1 below

Note 1: Revision E improvements vs v1.1 UL release

mbed-OS improvements:

- Addition of MAX32552 support & rework of file layout in target/ folder
- Non security related bug fixes in Maxim drivers and performance improvements.
- Rework of the linker file to adapt to separate box signature mechanism
- Some adjustments in the Maxim startup files

uVisor's improvements:

- Upgrade from 0.26.2-24 to 0.27
- Support for separate box signature (see related section in the main documentation)
- New API to get signing key of a box (hence it's privilege)
- Support for MAX32552
- Fix of Makefiles for correct multiple target support
- Improvement of debug messages in the debug version
- Add hook to implement ACL verification and enforcement at box loading time depending on signing key
- Add handling of NMI faults

Deeptrust API's improvements:

Globally:

- Support for separate box signature
- Reorganize folders
- Isolate crypto buffers
- Add key manager & crypto services
- Separation between core firmware, firmware level boxes, trusted boxes, and other boxes

PCI services:

- Addition of a watchdog to automatically clear the PIN if not used
- Forcibly flush the PIN after use
- Keep crypto working buffer private (used to be visible from several boxes)
- Erase buffers containing sensitive data after use: data_hash.data_val, issuer_public_key
- Other minor bug fixes

Secure Sandbox services:

- Add a trace capability to catch security issues, and capability to add user handling for such events
- Add secure RTC control
- Add automatic reset every 24h
- Evolutions in keypad handling and display following MAX32552 support addition
- Buffers containing keypad/touchscreen entries are now correctly kept as private
- Support of power management

Author

Maxim Integrated

Date

2016-2017

Reference

SPEC98T17

2. Copyright Notice

3. Trademarks

- ARM is a registered trademark and registered service mark and Cortex is a registered trademark of ARM Limited.
- mbed is a registered trademark of ARM Limited.
- All trademarks not mentioned here that appear on this web site are the property of their respective owners.

4. Introduction

This document describes the Deeptrust architecture and API.

Deeptrust is a security architecture designed by Maxim Integrated. This architecture allows software isolation and privilege limitation of software running on a System-on-Chip (SoC) (aka microcontroller) with an ARM® Cortex®-M3/4 core. This security design prevents applications hosted on the same platform from accessing sensitive data, peripherals, executable code.

Deeptrust has been designed to pass the Payment Card Industries – Pin Transaction Security 5.0 certification. This architecture aims to be approved as conforming to those requirements, so the customer will only need to build a conforming terminal (which must be evaluated) and have their changes to Deeptrust approved. The coverage of the requirements is exposed in the section PCI PTS POI 5.0 Guidance.

Software running in the platform is split into secure containers in order to:

- reduce the scope of the software validation
- reduce the scope of the PCI evaluation, and exclude non related code that can be modified more easily without undergoing a new evaluation
- improve the overall robustness of the running software by preventing the propagation of a bug or vulnerability

The Deeptrust offering contains the following:

- A Secure Boot ROM, embedded in MAX325xx ICs from Maxim Integrated
- Software sources and/or binary libraries that implement the security architecture mentioned above:
  - The lightweight hypervisor, derived from ARM uVisor (see `https://www.mbed.com/en/technologies/security/u`
  - ARM mbed® TLS (`https://tls.mbed.org`) leveraging the secure cryptographic algorithms
- Maxim software libraries sources and/or binaries that provide base services for customers to implement EMV-Level 2 and payment applications:
  - Secure Smart Card communication, display, user entry
  - Secure PIN handling (offline verification, DUKPT for online verification)
  - Mag Stripe reading
  - Secure cryptographic algorithms and key management
  - Secure memory allocation
  - Platform integrity management, error and security event handling
  - Secure firmware update
- General purpose software:

  - ARM mbed OS (using the mbed command line version, see https://docs.mbed.com/docs/mbed-os-handbook/en/2/getting_started/blinky_cli/)
  - Maxim MAX325xx HAL (drivers)
- Maxim PC Tools for Secure Boot ROM management:
  - generation of signed firmware images
  - secure update protocol
- Documentation:
  - The present user guide, including a description of the coverage of the PCI PTS POI 5.0 requirements
  - Secure Boot ROM documentation
  - PCI PTS Security Evaluation report to be reused for the final certification
- Demonstration and example code
- Technical support

5. Security Architecture Presentation

Deeptrust relies on the following elements of architecture:

- Hypervisor-based software isolation
- Chain-of-Trust, firmware integrity and authenticity
- Hardware enforced security
- Secure API

Note to the reader: the security mechanism presented here is composed of elements closely tight together and heavily depending on each other. Therefore it is advised to read this twice.

5.1 Primer on Cortex-M security mechanisms

Let's introduce some essential concepts now for the sake of understanding this document. These concepts will be further explained later though.

The architecture of Cortex-M3/4 cores provides the following security mechanisms:

- Two-level execution privilege: ”Thread user” and ”Handler privileged”. The core always boots in Handler privileged mode. Switching to Thread user mode is done via a dedicated instruction. Switching back to Handler privileged mode is achieved through Interrupts and exceptions (including the SVC software exception). Both modes can have separate stacks. Thread user also has limited access to core registers (MPU configuration, stack pointers, NVIC interrupt controller configuration, etc)
- Memory protection unit (MPU): this component is optional regarding the Cortex-M core instantiation, but mandatory for the security architecture presented here. The Maxim Integrated's MAX325xx SoCs do have an MPU implemented. The MPU allows to define access rules to any memory range (Flash, RAM, peripherals) for code running in Thread user mode. The MPU can be configured only from the Handler privileged mode of operation of the Cortex-M core.

5.2 Hypervisor-based software isolation

Software isolation is achieved by a lightweight security hypervisor (also simply called hypervisor below) that leverages the Cortex-M security mechanisms.

The lightweight security hypervisor in use is ARM® mbed uVisor (see [DOC1]). Software isolation means that the various software components run inside secure containers (aka ”boxes”) isolated from each other, and also from the hypervisor itself.

In this security architecture, the hypervisor is the only code running in Handler privileged mode of execution, and hence is fully and exclusively controlling the MPU.

5.2.1 Code partitioning: Core firmware and Secure containers (aka ”boxes”)

In this architecture, we distinguish the Core firmware from the Secure boxes.

- Core firmware: the core firmware contains the trusted executable code and is signed using the firmware signing key of the secure boot ROM (see), i.e.:
  - SoC Startup code
  - Hypervisor
  - mbed-OS (high level API, RTOS) and Maxim driver libraries
  - Other Maxim software libraries, C runtime library
  - Related constant data and variable initialization values
  - Store of Secure boxes verification keys

  The core firmware integrity and authenticity is verified and executed by the secure boot ROM (see Hypervisor initialization). This component is part of the PCI firmware perimeter.
- Secure boxes: Boxes contain application-level code. As seen above, secure boxes can be invoked by other boxes through the RPC mechanism. No direct call is allowed.
  - PCI Secure Services and Secure Sandbox Services boxes
  - Payment application box
  - Other secure boxes

Secure boxes are software containers enforced by the hypervisor. Thanks to the hypervisor, each box has:

- its own private stack
- its own private RAM
- its own private flash data section
- access to the smallest subset of peripherals possible as defined by access control lists (ACLs). ACLs are used to deny/grant physical access to peripherals.
- an ID and a namespace that can be trusted. Those IDs are used for granting or denying access during RPC calls (described later).
- no access to forbidden peripherals, memory ranges, exception vectors, NVIC configuration, MPU configuration

This way, even if all are running with unprivileged permissions, different boxes can protect their own secrets and execute critical code securely.

Note that the hypervisor allows the existence of a ”Public box”. In this design, the public box is part of the ”Core firmware” and is actually not executing code. It is not at the same level as Secure Boxes described below.

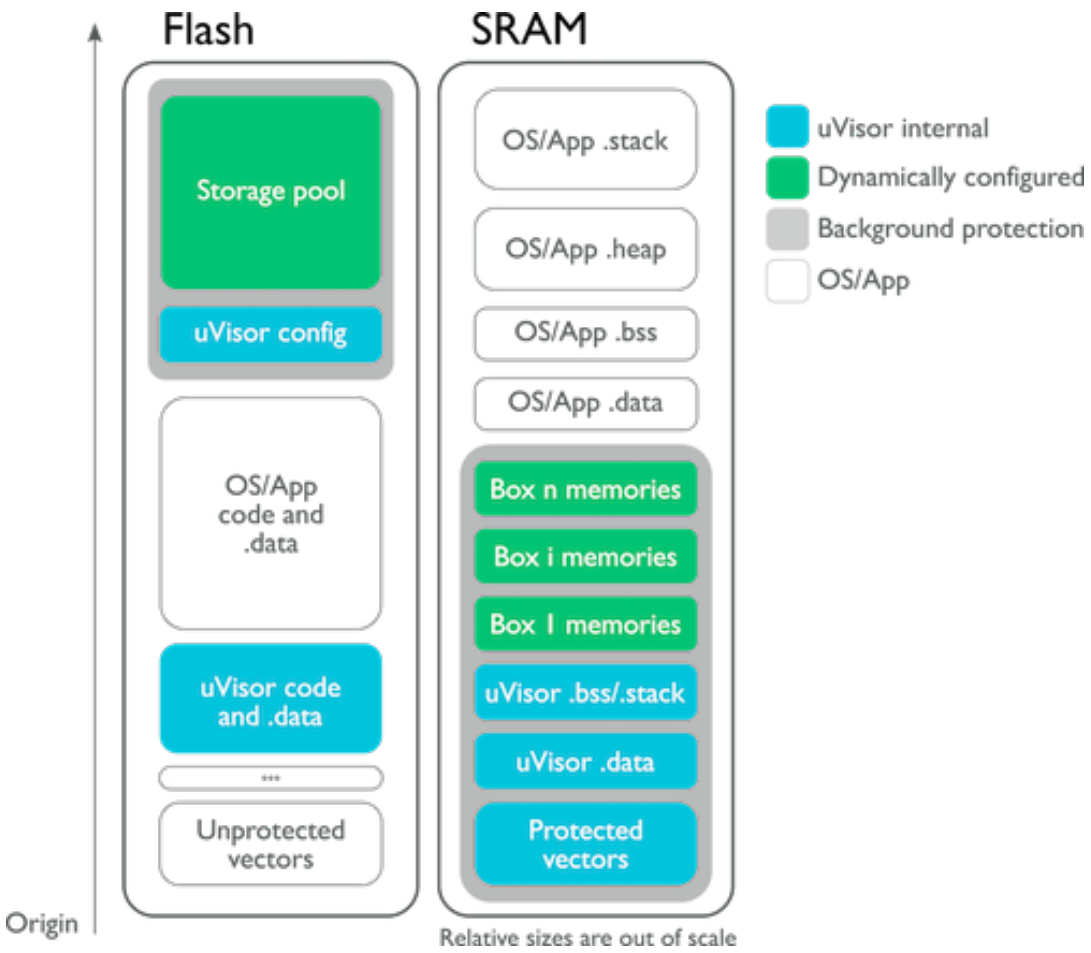


Figure - Platform memory layout

There are three levels of secure box privilege levels:

- ”Box firmware”: the secure box is part of the PCI firmware perimeter; it has full access to peripherals and memory(∗)
- ”Box trusted”: the secure box is trusted, which grants some direct access to peripherals and memory
- ”Box trusted”: the secure box is not trusted, which grants no direct access to hardware and limited visibility of memory

(∗) the hypervisor still has protected memory that is not visible from Thread/user mode, hence from no box whatever the box privileges are.

The box privilege is granted at box loading time based on the signing key of the box. If the signing key was K_fw and the signature verification of the box is successful, the the ”Box firmware” privilege is granted.

This box privilege level defines two kinds of privileges:

- The allowed ACLs
- The allowed services

The box privilege policy can be customized:

- from the core firmware, it is possible to define what ACLs are acceptable for each box privilege level. The ACLs of secure boxes cannot be arbitrary and must correspond to what is really allowed by the core firmware. Therefore the hypervisor is in charge of controling the validity of the ACLs to make sure that memory regions that need to be always protected are not claimed by the secure box.This specification defines a sample ACL policy, but it can be adapted. This policy is implemented in the core firmware, thus cannot be tampered with by additional software. The ACL policy is enforced at boot time when secure boxes are laoded by the hypervisor. It cannot be modified at runtime, as for the ACLs themselves.
- from boxes that offer services through RPC. The hypervisor offers a service to be used by RPC servers that allows to get the privilege level of the caller box in order to implement an access policy based on the caller's privilege. For instance, access to the ”PIN entry” service can be declined to non trusted boxes. In addition, the box name can be used to further refine the policy (e.g. service is allowed to trusted boxes which name is box_aaaaa and box_bbbbb, but declined in all other cases). The box name can also be asked by the RPC server, similarly to the privilege level of the caller.

5.2.1.0.1 Box access control list

For secure boxes, unprivileged access to selected hardware peripherals and memories must be explicitly granted through Access Control Lists (ACLs). ACLs are managed by the hypervisor. ACLs are defined at boot time and cannot be modified at runtime. When running out of a secure box, access to memories and peripherals is the most restrictive: access that are not explicitly granted are denied by default.

Software running in a box can access only the memory ranges it has been allowed to (through the Access Control Lists, ACLs), with the access type enforced (RWX, RO, etc...) for each range. The definition of the privileges of a box is done at compilation and remains static, i.e. cannot be modified at runtime.

In addition, the secure box binary code, constant data and variable initialiation values and its ACL are signed as a whole (single binary image) using a private signing key. As the whole image signature is verified upon boot, the box privileges and ACLs are protected against modification. The application's box privileges are read by the hypervisor during the early initialization of the SoC (see Hypervisor initialization) and stored into the hypervisor protected memory so they cannot be modified from the application's box.

5.2.1.0.2 Box isolation enforcement

Let's explain how the hypervisor implements the isolation of secure boxes. The hypervisor is the only software running in Handler privileged mode. All other software runs in Thread User mode (see Cortex-M mechanisms: MPU, privileges, NVIC), even the core firmware's unprivileged code.

The hypervisor is therefore the only software component allowed to control the MPU (initial configuration, re-configuration during context switches, secure interrupt handling) and catch the exceptions and interrupts. This is verified as:

- the hypervisor executable image is verified by the Secure Boot ROM (see Secure Boot firmware and verification key are in immutable memories with integrity check), hence it cannot be modified arbitrarily.
- the hypervisor is initialized early in the boot process (see Hypervisor initialization), before any other Secure Box can run. During its initialization, the hypervisor:
  - registers the exception vectors, and prevents access to the vector table and to the vector table location
  - sets the core in Thread User mode of execution before relinquishing the CPU to the applications.

The only way to jump back to Handler privileged mode is through exceptions (interrupts and exception, SVC instruction). As exception vectors belong to the hypervisor, the hypervisor keeps the monopoly of execution in Handler Privileged mode.

The hypervisor relies on the two hardware mechanisms described hereafter (see Cortex-M mechanisms: MPU, privileges, NVIC).

5.2.1.0.3 Instantiation of secure boxes and code signing

Secure boxes cannot be instantiated at runtime. Instantiation is done at compilation time, as the definition of the ACLs, using the hypervisor API. Once compiled, the binary resulting of the build process contains the application executable image, containing the ACLs. This image must be signed so it can be part of the 2nd level application.

The file `sources/deeptrust_api/box_example/example.cpp` illustrates this instantiation.

In the first version os this specification, the compilation of the whole embedded software must be done in a single operation. In the resulting binary image, the core firmware binary is separated and signed separately using the ROM secure boot signing key. Other secure boxes code is also separated from each other and signed with different signing keys. However the software binary image remains monolithic and must be loaded entirely in a single operation. Secure boxes are stored in a contiguous memory area.

In the next release, it will be possible to compile secure boxes separately, and load them separately from each other.

Secure boxes have the following structure in memory:

- Header with:
  - 4-byte magic
  - Box name (must be unique, update fails if box name already exists)
  - Box image length
  - Box verification key ID
  - Box ACLs, RPC list
  - Signature of the above field and the ”Secure box executable image” below, using the verification key ”ID” as mentioned in the header.
- Secure box executable image
  - Executable code (.text section)
  - Constants and initialization data (.rodata and .data sections)

Note: The box name must be unique. Note: The above is signed using a dedicated key that is different from the CRK core firmware signing key. This scheme allows to consider secure boxes as separate from the core firmware.

The verification key ID defines the box privileges. If one posseses the K_fw signing key, he can write code having the ”box firmware” privilege. The K_trusted shall be distributed

Even if no unsigned code can be executed (this leads to a platform reset), any code must be carefully reviewed before being signed and loaded into the platform.

Let's precise that the code signing key management is left at the discretion of users of Deeptrust. The foreseen usage is:

- The terminal vendor owns the CRK and the K_fw keys that allow writing code within the ”PCI perimeter”
- The terminal vendor or a contractor owns the K_trusted key taht allows writing ”Banking applications” allowed to somehow interact with smart cards and magnetic stripe (through usage of the PCI secure services)
- Other parties may own the K_other key to be able to provide low privilege applications (advertisement, loyalty...). The K_other key is designed to give the least privilege in the platform.

This scheme can be further extended with more keys, and customized by reworking the `uvisor_check_acl` function. Ultimately the vendor is in charge of managing the code verification keys present in the platform, an how to distribute/update them.

The Makefile present at the root of the project is in charge of signing secure boxes according to this 3-level privilege design.

Notes:

- There's no 1 to 1 mapping between boxes and threads: a box can run several threads, all running with the parent box privileges.
- The program entry point ”main()” runs outside of a secure box (All the code running in Thread User mode, that is not protected in a secure box, is referred to as the public box). this public box is part of the ”core firmware” and cannot be customized unless one posseses the ROM secure boot signing key.

5.2.2 Hypervisor initialization

During the boot of the SoC, the Secure Boot ROM starts running in ”Handler/Privileged” mode (the Cortex-M boots in that mode). It verifies the digital signature of the ”2nd level application” stored in flash memory, and, if successful, jumps into the entry point (aka startup) of the latter, also in ”Handler/Privileged” mode.

The the ”2nd level application” is a term used in the Secure Boot ROM documentation. It designates the binary executable image that sits in a flash memory and that is verified by the Secure Boot ROM. In our design, the ”2nd level application” is the ”core firmware” described above.

The ”core firmware” execution begins by the initialization of the hypervisor itself, which also runs in Handler/↩ Privileged mode. During this initialization phase, the hypervisor sets up a protected environment using the M↩ PU: the MPU is configured so that the hypervisor keeps ownership of its own memories and the security-critical peripherals, in order to keep them protected from the unprivileged code. The hypervisor secures two main memory blocks, in flash and SRAM respectively (it places its own constants, data and stack in secured areas of memory, separated from the unprivileged code). In both cases, it protects its own data and the data of the secure boxes it manages from the unprivileged code.

This initialization step makes some peripherals impossible to access to the unprivileged code whatever the AC↩ Ls are. Therefore accessing to some security-critical peripherals (like DMA) requires SVCall-based APIs for the unprivileged code. The MPU configuration, the vector table, the NVIC configuration are also made inaccessible from the unprivileged code.

The hypervisor initialization is as follows:

1. Several sanity checks are performed, to verify integrity of the memory structure as expected by the hypervisor.
2. The hypervisor ”.bss” section is zeroized, the data section initialized.
3. The Memory Protection Unit (MPU) is configured
   - The hypervisor takes ownership of the vector table
   - The hypervisor protects some RAM and some flash so they are reserved to Handler privileged mode
4. Secure boxes are loaded:
   - Secure boxes digital signatures are verified one by one using the appropriate verification key
   - It is verified that names of secure boxes are unique
   - Each box's .bss section is zeroed
   - Each box's .data section is initialized
   - Access Control Lists (ACLs) are registered and checked against validity. Additional boxes cannot be granted arbitrary access to any memory region (the ACL policy described earlier is enforced)
   - Stacks are initialized, a private box context is initialized, if required by the box.
5. Handler privileged and Thread user modes stack pointers are initialized.
6. Execution switched to Thread user mode and handed over to the unprivileged code.

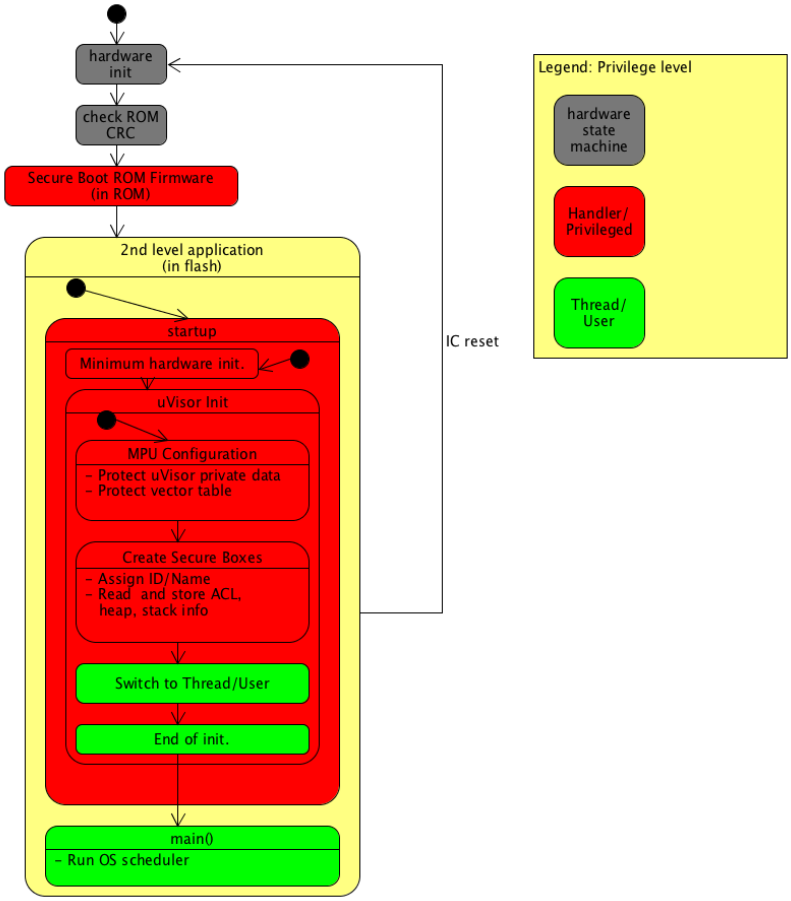


Figure - Boot sequence

5.2.3 Context switches

The hypervisor prevents CPU registers leakage when switching execution between Handler privileged and Thread user code and between mutually untrusted unprivileged boxes. It also remaps the stack and modifies the MPU according to the destination context.

During a context switch, the hypervisor stores the state of the previous context and then:

- It re-configures the stack pointer and the box context pointer.
- It re-configures the MPU and the peripherals protection.
- It hands the execution to the target context.
- A context switch is triggered automatically every time the target of a function call or exception handling routine (interrupts) belongs to a different secure box. This applies to user interrupt service routines, threads and direct function calls.

Memory map varies according to context. The diagram below shows the modification of the memory map when switching from a Box A to a box B

Memory Map when running in context A
Peripheral Display : DENIED
Peripheral KBD: DENIED
Peripheral SmartCard: DENIED
Other Peripheral
Private RAM of Context A
Private RAM of Context B: DENIED
Other RAM
Flash

Memory Map when running in context B
Peripheral Display : OK
Peripheral KBD: OK
Peripheral SmartCard: OK
Other Peripheral
Private RAM of Context A: DENIED
Private RAM of Context B
Other RAM
Flash

5.2.3.0.1 Initial jump to Thread user mode

All the code that is not explicitly part of the hypervisor is generally referred to as unprivileged code. The unprivileged code is the code that runs in Thread/User mode, and that is therefore unable to modify neither the configuration of the MPU (privileged access required) nor its own access privileges.

The ”main()” function of the 2nd level application, called from the startup file, after the initialization of the hypervisor and the C/C++ runtime, is such unprivileged code. Indeed, after initialization, the hypervisor starts executing the ”main()” after having switched into the Thread/User mode of execution.

Unprivileged code runs with the following capabilities:

- runs in Thread/User mode
- has direct memory access to unrestricted unprivileged peripherals (as defined by the Cortex-M)
- can require exclusive access to memories and peripherals
- can register for unprivileged interrupts
- cannot access privileged memories and peripherals (doing so makes the MPU trigger a CPU fault)

5.2.3.0.2 Interrupts and exceptions

Exceptions/interrupts make the Cortex-M switch to Handler privileged mode. Therefore the hypervisor must catch the exception, switch to Thread/User mode and call the registered unprivileged handler.

To this end, interrupt vectors are relocated to the SRAM but protected by the hypervisor. Access to them is made through specific APIs: the unprivileged code can register for unprivileged interrupts. Therefore the hypervisor needs to catch, forward and de-privilege interrupts to the unprivileged handler that has been registered for them.

A context switch is triggered automatically every time an exception handling routine (interrupts) belongs to a different secure box.

5.2.3.0.3 Cortex-M Privilege escalation

Interaction from the Thread user code to the Handler privileged code is achieved by exposing SVCall-based APIs. As exception vectors are handled exclusively by the hypervisor, SVC exceptions are caught by this one exclusively, as all the other forms of exceptions (and interrupts).

5.2.3.0.4 RPC mechanism

The hypervisor allows inter-box communication through RPCs (Remote Procedure Calls). The callee box can check the origin of the call using the box ID of the caller (and hence it's unique name), and also the box privilege of the caller, in order to determine whether the requested action is legitimate or not. RPC calls trigger context switches.

Definitions:

- Context A: all applications (boxes) Their ACLs prevent direct access (hardware registers) to sensitive peripherals
- Context B: PCI box, Secure services box Their ACLs allow direct access to sensitive peripherals

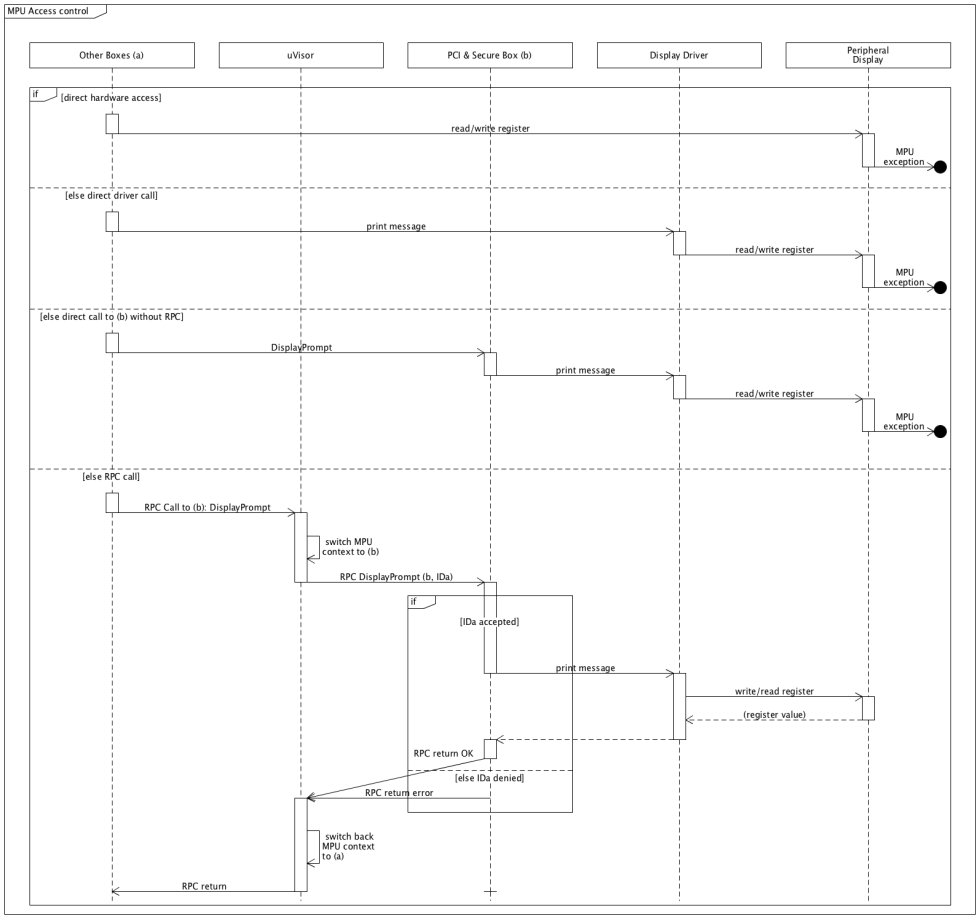
Security context switch (from Box A context to box B context) is done via RPC calls

1. SVC instruction triggers exception caught by the hypervisor
2. RPC is managed by the hypervisor
   (a) Box A calls RPC (ARM SVC instruction)
   (b) Exception triggered by SVC: enters into Handler privileged mode and enters the hypervisor's "software exception" handler
   (c) the hypervisor: Gets caller box (A) ID, target box (B) ID
   (d) the hypervisor reconfigures the MPU with context B ACLs
   (e) the hypervisor: exits Handler privileged mode, dispatches call to target box
   (f) Box B: executes called function

On return, inverse process is applied.

Context switches can also be triggered by interrupts also handled by the hypervisor, the same way. An interrupt makes the Cortex-M enter the Handler privileged mode. However exception vectors are protected by the hypervisor hence the hypervisor catches the interrupt. The execution mode of the Cortex-M is set back to unprivileged by the hypervisor and it then calls the registered interrupt handler.

5.2.4 Summary of MPU protection effects



5.3 Chain-of-Trust, firmware integrity and authenticity

5.3.1 Secure Boot firmware and verification key are in immutable memories with integrity check

The secure Boot ROM is a trusted immutable code in read-only memory that allows booting the SoC. The use of a Read-Only Memory guarantees de-facto the integrity and authenticity of the Secure Boot firmware itself, as modification of the ROM on an IC die is assumed too costly and highly technical. Integrity of the ROM is verified by a CRC verification before boot: the hardware initialization state machine verifies the CRC before releasing the CPU upon each reset. The CPU will start executing the code only if the ROM CRC is correct.

The Secure Boot ROM guarantees that the firmware that gets executed next is genuine and not modified (authenticity and integrity verification). This is mandated for building trust in the system. The Secure Boot ROM is the Root of Trust.

The secure Boot ROM is designed to launch a 2nd level application from the internal flash memory of the SoC. It requires that the 2nd level application's digital signature is correct and has been performed with the correct CR↩ K_sign signing key, otherwise it refuses to start it. This digital signature is verified using a public key (Customer Root Key, CRK_verif) stored in a write once memory (OTP, one-time programmable memory) of the SoC.

The CRK_verif integrity is also verified before use. Indeed the MAX325xx contains a public key in ROM (the MRK, Maxim Root Key), owned by Maxim Integrated. The MRK allows verifying the signature of the customer

public key (CRK_verif) before it is used. Therefore, customers must submit their public key CRK_verif to Maxim beforehand. Maxim signs this public key with the MRK private counterpart, and returns this signed CRK_verif to the customer. The customer then has to download this signed CRK_verif through the above-mentioned SCP protocol (SCP packets are pre-generated offline in a secure environment using a private key see [DOC11])

As the secure boot ROM firmware and the CRK_verif and MRK verification keys cannot be replaced, the integrity and authenticity of the booted 2nd level application is guaranteed.

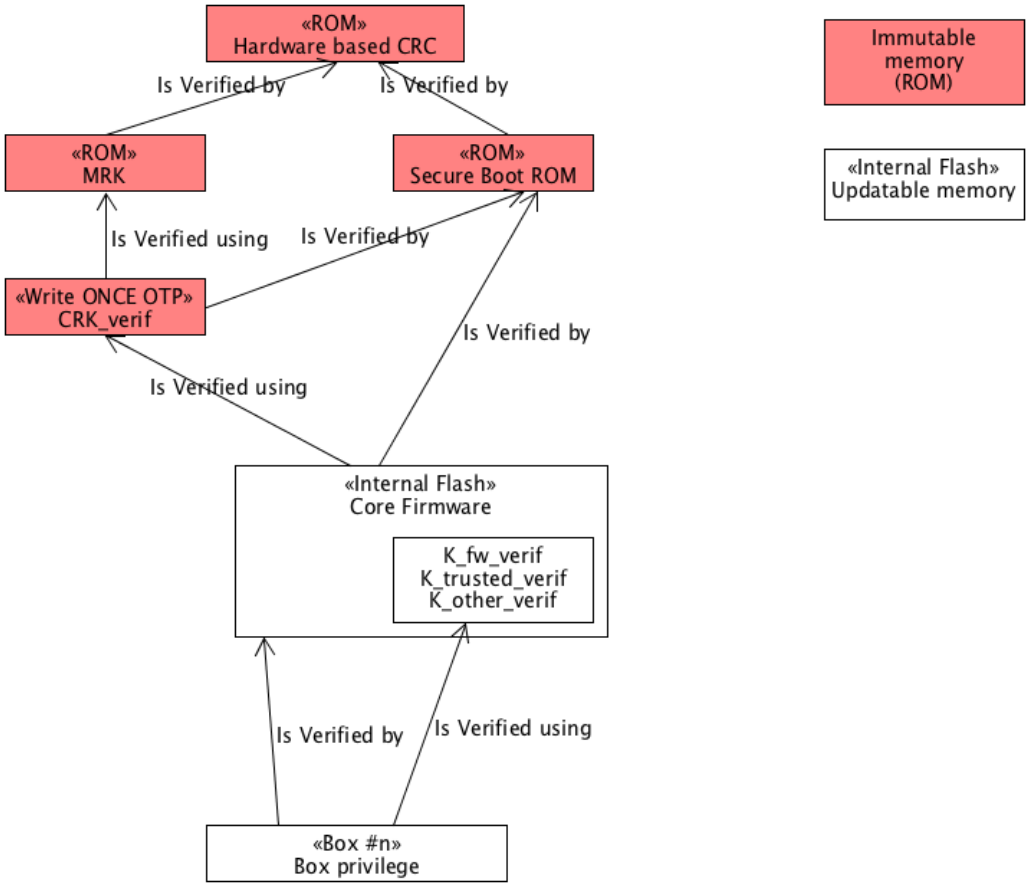


Figure - Chain of trust

5.3.2 No code loading/injection is possible except through a secure loader

The below items guarantee that only trusted code can be loaded into the platform.

5.3.2.0.1 Signature verification of downloaded code

The default ACL policy does not grant write access to the internal flash to store executable code and therefore does not allow code updates. However the Secure Boot ROM is capable of updating applications through a serial protocol called SCP. It can securely download new applications into the platform's non-volatile memory through a serial port or a USB port.

Note: The SCP protocol also allows configuration, life-cycle management and key management. See [DOC11]

5.3.2.0.2 Loading and Update of additional secure boxes

In the second version of this specification, a partial update service will be offered, where secure boxes can be independently loaded and/or updated. Those secure boxes must be signed, and the verification will be performed during the uploading (in addition to theit boot time verification already described). An alternate loading service will be provided too, in order to second the SCP offered by the ROM.

To allow partial build, the core firmware will publish the location of symbols so that additional boxes can refer to those symbols (e.g peripheral drivers entry points, C library functions). In order to be able to update the core firmware without breaking the compatibility of the additional boxes already present in the platform's flash memory, those adresses must be kept the same. Some indirect calls can be used to make the preservation of constant adresses easier. Note that an update of the core firmware will most of the time require a recompilation and an update of all secure boxes. Otherwise, secure boxes will possibly be modified, built and loaded separately without any impact on the rest of the software.

Secure boxes will have to be built using position independent code so that they can be stored at random locations. The volatile memory needed by the them will also have to be dynamically allocated, because the precense of other secure boxes using potentially the same otherwise static volatile memory adresses may conflict.

Interfacing secure boxes

Secure boxes cannot contain general purpose code accessible through direct call. As a matter of fact, existing secure boxes are usually not aware of newly added boxes and therefore cannot really use them. The boxes can however publish services in a jump table. Those services are RPC functions. Additional boxes can use services by other boxes already present in the platform's flash memory. They can refer to these services knowing the targeted box's name and service ID.

5.4 Summary of software items, keys and their protection

Software item	**PCI Perimeter?**	**Execution privilege**	**Signing key**	**Verification key**	**Location of verification key**	**Verification of verification key by:**	**Verification of software item by:**
ROM secure boot	Yes	Handler privileged	None		n/a	n/a	Hardware state machine performs CRC-32 of ROM
Core firmware - privileged (hypervisor)	Yes	Handler privileged	CRK_sign	CRK_verif	OTP memory	ROM secure boot, using a hard-coded MRK public key in ROM	ROM secure boot
Secure box - ”box firmware” privilege	Yes	Thread user	K_fw	K_fw_verif	In core firmware binary image	ROM secure boot, together with the Core firmware image	Core firmware's hypervisor

Software item	PCI Perimeter?	Execution privilege	Signing key	Verification key	Location of verification key	Verification of verification key by:	Verification of software item by:
Secure box - "box trusted" privilege	No	Thread user	K_trusted	K_↩ trusted_↩ verif	In core firmware binary image	ROM secure boot, together with the Core firmware image	Core firmware's hypervisor
Secure box - "box other" privilege	No	Thread user	K_other	K_otehr_↩ verif	In core firmware binary image	ROM secure boot, together with the Core firmware image	Core firmware's hypervisor

5.5 Additional considerations

The following paragraphs give additional justifications to the consistence of the above, security-wise

5.5.1 Absence of backdoors

All MAX325xx SoCs released in the field have no JTAG interface enabled. Therefore it is not possible to bypass or perturbate the executed firmware (in particular the secure boot ROM), or to inject arbitrary code for execution by the platform through a debug interface.

No test mode can be activated after manufacturing operations at Maxim Integrated therefore no possibility to bypass security is provided.

No intentional backdoors for re-enabling the JTAG or any test mode are existing.

The Secure Boot ROM is always executed after reset. It cannot be skipped.

5.5.2 Execution from internal memories

The platform software can be executed from internal flash and optionally internal RAM, and the runtime data are stored in internal RAM. Long term data may be stored in internal flash. Access to the SoC internal memories is a costly attack defeated by the presence of a die shield and is assumed impossible in the present context.

5.5.3 Protection of external memories

The platform software executed from external flash and the related data are protected by the on-the-fly integrity and encryption, and the external flash is protected from physical tampering/replacement throught the SoC intrusion sensors that prevent intrusions in the enclosure. Therefore the external flash memory is protected against manipulation.

Therefore, the only mean to load executable code into the platform is through the Secure Update provided by the Secure Boot ROM.

5.5.4 Early execution the Secure Boot ROM and of the isolation mechanism

The SoC always boots the Secure Boot ROM in any circumstances and starts the 2nd level application (if valid, as discussed above).

The 2nd level application begins by the early initialization of the SoC, the initialization of the security sensors of the SoC and then the initialization of the the lightweight security hypervisor.

The hypervisor is initialized right after device start-up and takes ownership of its most critical assets, like privileged peripherals, the vector table and memory management.

From this moment on, the operating system/application runs in Thread user mode and in the default context, which is the one of the main box.

The next step consists in the creation of the various boxes, the registration of their ACLs and the beginning of the execution of the OS Scheduler that will execute the various threads (a box may contain one or more threads). The overall code is covered by the Secure Boot ROM integrity and authenticity verification and can therefore be trusted.

Additional firmware independent components can be verified and launched by the initial trusted firmware already verified above.

5.6 Hardware enforced security

The following hardware features are leveraged by the software described above.

5.6.1 Cortex-M mechanisms: MPU, privileges, NVIC

The Cortex-M3/4's Memory Protection Unit and the Execution mode/Privilege level (Thread/Handler modes of execution, Privileged/User Access) are used to support the software isolation.

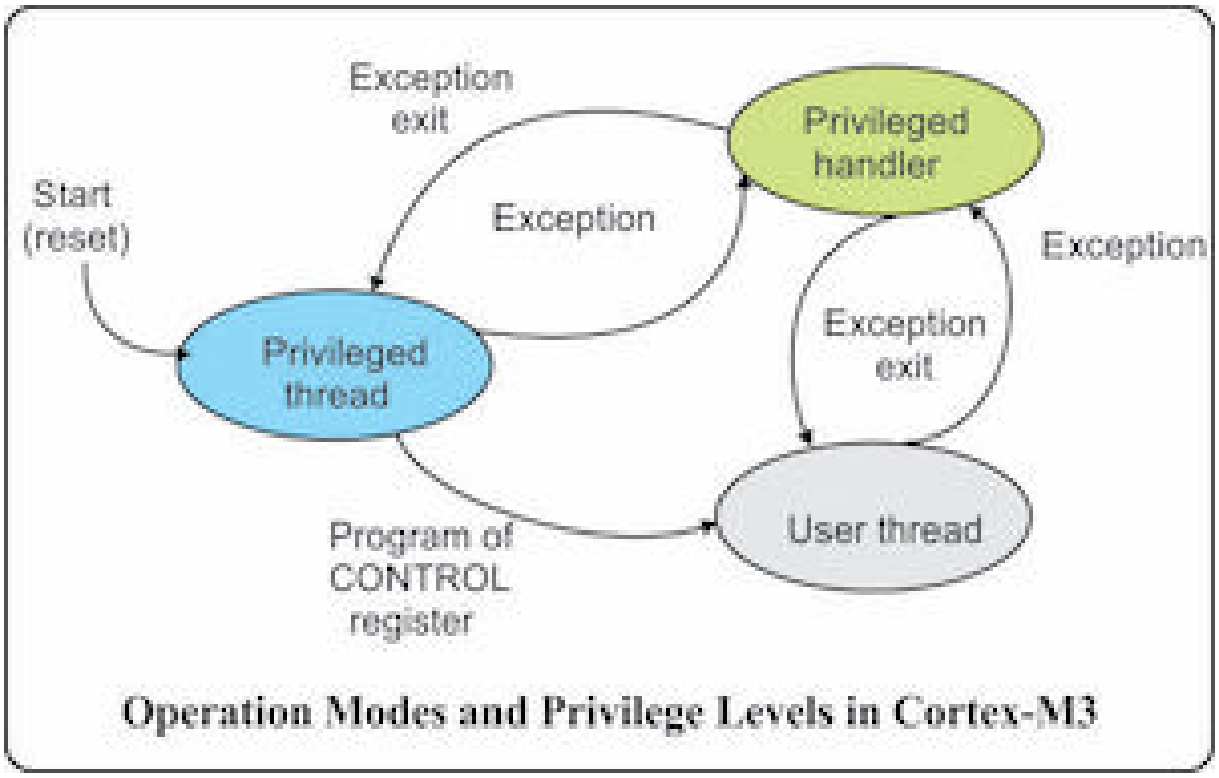
5.6.1.0.1 1. The Cortex-M's Memory Protection Unit (MPU)

The Cortex-M MPU is used to grant/deny access to regions of the memory map to the unprivileged code (i.e. all the code except the hypervisor)

The MPU could also restrict access even to the privileged code (without the possibility for the privileged code to modify the access) but this feature is not used currently, in the security architecture presented here.

5.6.1.0.2 2. The Cortex-M's 2-level execution model

The Cortex-M's 2-level execution mode (Handler, Thread) combined with the 2-level privilege system (Privileged, User) prevents code running in Thread/User mode from breaking the MPU configuration that requires Privileged access. This hardware feature is leveraged by the lightweight security hypervisor. Return to the Privileged/Thread mode can be done only via a SVCall-based API (see http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0179b/ar01s02s07.html), i.e. a software triggered exception that can be cached only by the Privileged Handler actually implemented by the Lightweight security hypervisor and that cannot be modified by/re-routed to unprivileged code.



5.6.1.0.3 3. The Cortex-M's NVIC

The interrupt controller is exclusively controlled by the hypervisor, hence no other unprivileged code can register exception handlers. This is vital since the Cortex-M executes exception handlers in Privileged/Handler mode.

5.6.2 NVSRAM - Battery backed non volatile RAM

Sensitive data can be stored in the MAX325xx's battery backed RAM which gets immediately erased in case of tampering detection (physical tampering, environmental perturbations). The NVSRAM is leveraged by the Key Manager to store long-term secret/private keys that get erased whenever the device is under attack.

5.6.3 Sensors

Environment perturbations (temperature, glitches, die shield, external dynamic sensors,...) lead to the generation of a non maskable interrupt that stops the execution of the platform's software and thus prevents running away into a non secure mode. Sensors are activated before the security hypervisor initialization in order to guarantee the physical integrity of the platform before running further code.

5.6.4 Read-Only Memory, One-Time programmable memory

The Secure Boot ROM is non modifiable and thus inherently trusted, and its configuration or the code verification keys are in a write-once memory (OTP) and cannot be changed. The ROM and the OTP memory areas integrity is verified (Checksum) before use.

Note: hardware security mechanisms are evaluated in a separate report. See report [DOC12].

5.7 Secure API

Access to PCI PTS POI sensitive services via API with security policy.

The services offered by the Maxim software library are contained in 3 boxes and can be invoked via RPC calls as described above.

The API entry points are implemented following this behaviour:

- get the ID of the calling context
- get box privilege of the calling context
- grant/deny access to the service depending on the caller's ID, the box privilege, and the context

3 boxes are offered:

- PCI Security Services boxes (In certification scope)
- Secure Sandbox Services (In certification scope)
- Security Monitor Services (In certification scope)

Additional services such as OPEN protocols would also fit into the Secure Sandbox Services.

The API can be found further in this document, see `Reference Guide`

5.8 Development process

5.8.1 Source code control

The source code of the whole Deeptrust software offer is stored on a secure server at Maxim Integrated. This server features access control:

- only well-known project stakeholders can read the source code
- only software developers and push additional code to the source code repository
- only the software integrator can merge-in new code from software developers
- only the software integrator can publish new releases after review

The list of stakeholders is controlled by the project manager and can be also modified by some IT administrators. All are personel with a high level of confidence. Roles are:

- simple stakeholder: read-only access
- developer: read access, write access to separate branches, cannot merge code into main trunk
- validator: read access
- integrator: read/write access + publication rights
- project manager: read/write access + stakeholder list definition

The source control is implemented using GIT, more precisely a self-hosted Gitlab server. The server is located on Maxim's internal network with all the usual protections applied to corporate level servers that contain company vital data.

5.8.2 Bug tracking

Bugs are tracked using Gitlab's bug tracker. Bugs can be reported by any stakeholder. They can be closed by the integrator or the project manager.

5.8.3 Code review

The following code review process is applied:

- The third party code and proprietary code are carefully reviewed by software security experts at Maxim, and submitted to an external lab for audit.
- Issue fixes are reviewed before merging in the main trunk of the code
- The integrator and project manager are in charge of ensuring reviews are done.

5.8.4 Source code control

The public repositories containing the third party source code included in Deeptrust offer is checked weekly for security bugs and vulnerabilities, in addition to a subscription to the bug trackers:

- https://github.com/ARMmbed/uvisor
- https://github.com/ARMmbed/mbed-os
- https://github.com/ARMmbed/mbed-cli
- https://gcc.gnu.org/bugzilla/buglist.cgi?chfield=%5BBug%20creation%5D&chfieldfrom=24h

Such issues are reported in Maxim's internal Bug tracking system for immediate analysis and correction if they are found exploitable. In case of exploitable vulnerabilities, new software releases are published to customers as fast as possible together with the notice describing the issue.

5.8.5 Developer's guidelines

In order to prevent the introduction of vulnerabilites, Maxim Software developers follow company rules for secure coding. These rules can be provided upon request.

In addition, static analysis is performed on the source code to detect potential security issues (RATS, CPPCheck, Visual Code Grepper are all used).

A thorough validation is also performed using the following strategy:

- development and review of a software test plan
- functional validation
- security validation: focus on detection of buffer and integer overflows, fuzzing of the APIs presented in this document, logical security bugs
- other validation (performance, documentation accuracy)
- delivery of a software test report The whole code provided in the Deeptrust offer undergoes the above rules.

5.8.6 Firmware versioning and management

The firmware is versioned following this scheme: vX_Y_Z

- X: is the revision major, currently "1" to match this specification. The mentionned version 2 of this specification will lead to an incrementation of this number.
- Y: is the revision minor. It is incremented every time new functionalities or bug fixes are done, leading to possible backward compatilibity issues may arise.
- Z: is the patch number. It is incremented every time a release is done. It usually corresponds to bug fixes or minor functional additions without compatibility issues.

The firmware is developed internally at Maxim using the source control, bug tracking and development guidelines described above. Reviews are performed and recorded on a regular basis, at least before each customer delivery. Validation is performed and recorded, at least before each customer delivery.

Firmware deliveries are tagged and documented (description of the list of changes), together with the review and validation records.

Firmware is released to customers using a public GIT server owned by Maxim, with strict read-only access and access control: Only customers being granted access can read the source code. The hypervisor is released in binary format only. The binary hypervisor is released in the mbed-os source tree, and included into the final software during the mbed-os build step.

5.9 Conclusion

The above architecture provides a strong isolation of the sensitive data from regular applications. Sensitive data are handled within Secure Boxes only. Other boxes cannot get access to the data manipulated by these boxes as well as to some peripherals of the platform.

Isolation is enforced by the lightweight security hypervisor, which code and configuration data are protected by the MPU and the execution mode/privilege level system of the Cortex-M core and verified by the ROM based secure boot. the lightweight security hypervisor is executed early in the startup sequence before any user code, and no debug/test/executable code loading exist in the platform.

Therefore the software isolation mechanism cannot be bypassed.

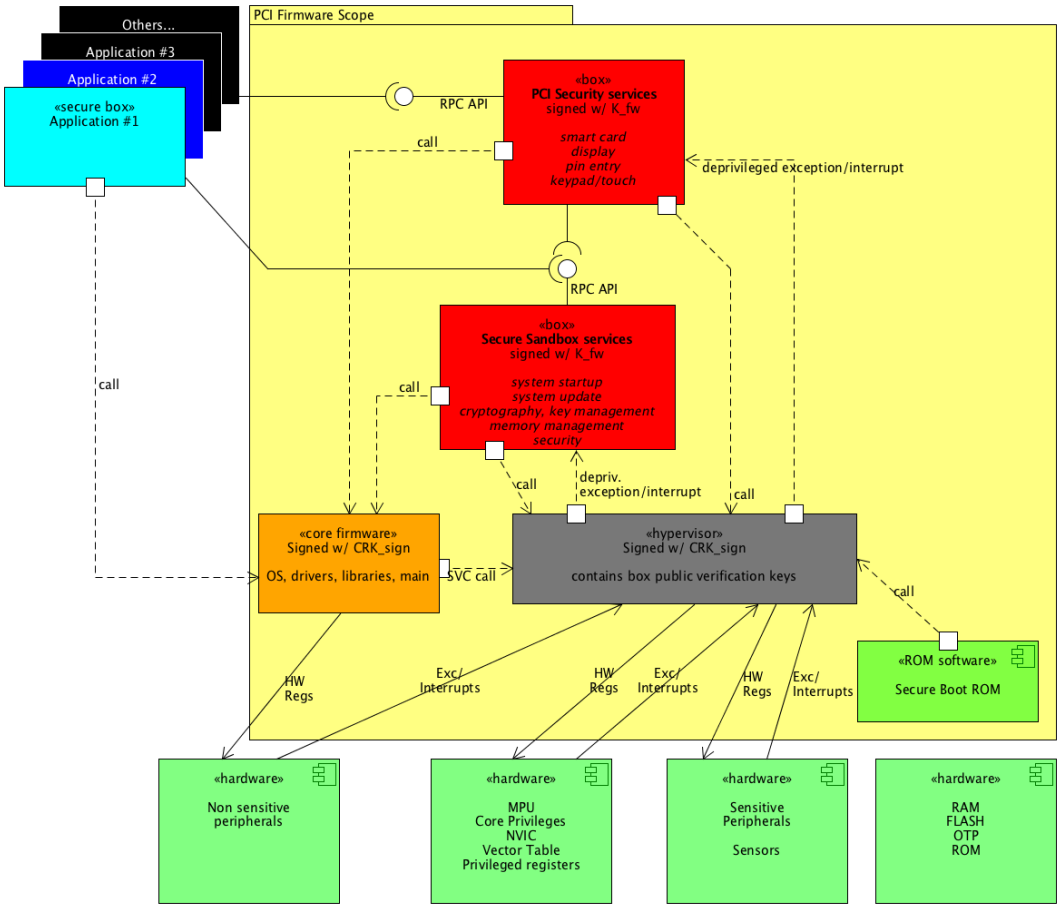
5.9.1 Architecture diagram, PCI firmware perimeter

The overall architecture relies on the following elements

- Application/other boxes (Out of PCI Certification)
- PCI Security Services boxes (In certification scope)
- Secure Sandbox Services (In certification scope)
- Security Monitor Services (In certification scope)
- Operating System (In certification scope)
- Lightweight Security Hypervisor (In certification scope)

- Secure BOOT ROM (In certification scope)

The architecture enforces the least privilege principle.



6. Design Description

6.1 Software API Specification

Usage reference:

6.2 Application box(es)

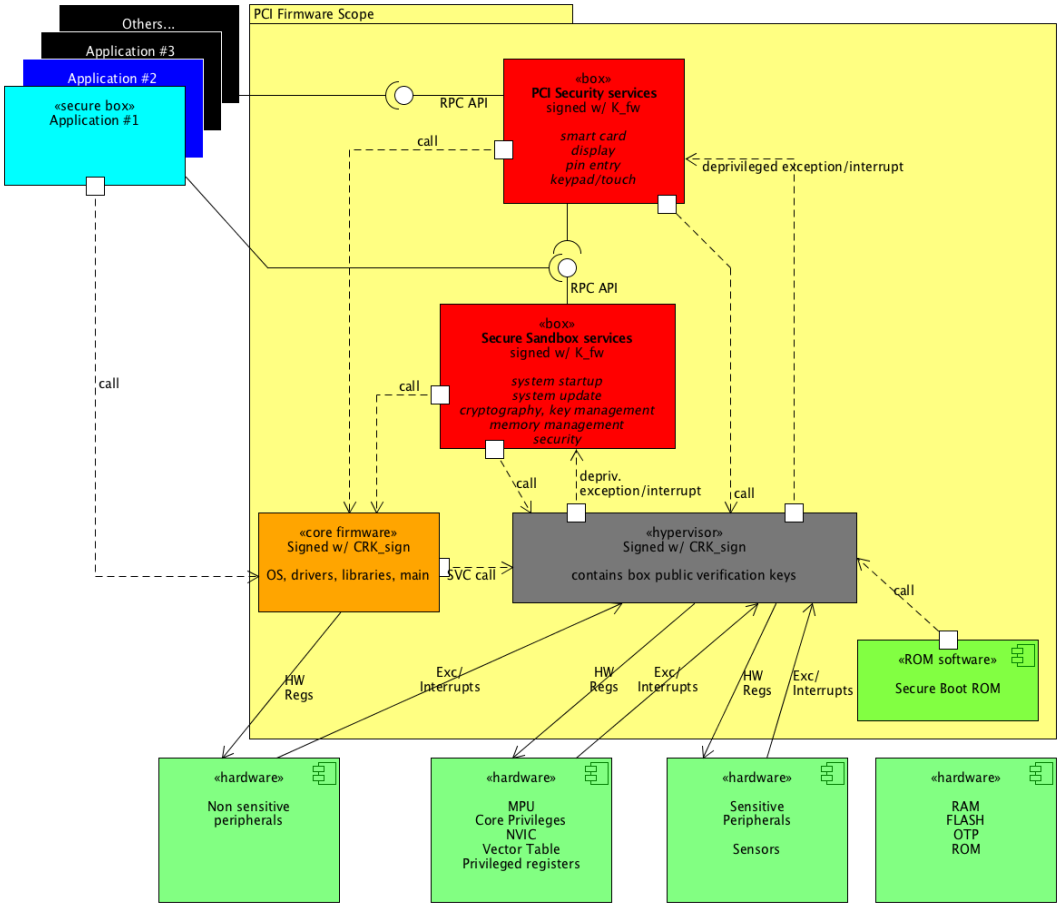
One or more boxes that run applications (e.g. VISA, EMV, Mastercard, Main application, etc) may exist and leverage RPC API of other boxes (Boxes can communicate with each other through Remote Procedure Calls aka RPC).

Their identification (box ID) will grant them access to their private data or specific privileges when using the API.

Those boxes run in independent threads.

PRIVILEGE LEVEL: ”Box Trusted” or ”Box Other”

They actually implement the high-level services proposed by the device (e.g. payment application). By using the Secure Sandbox services box and the PCI Security Services box, the applications may be kept out of the certification perimeter. Code can also run out of any box.



6.3 Operating system, drivers, C library, other libraries...

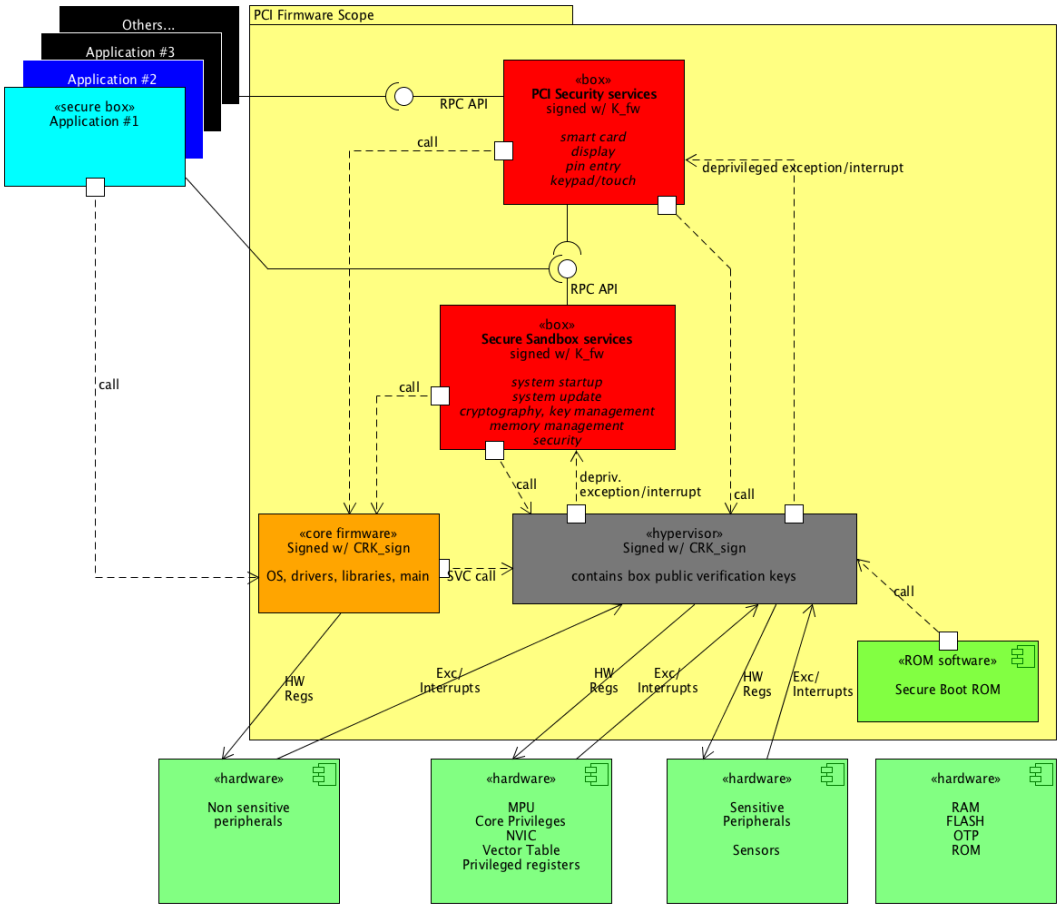
The following software items are not running in a separate box:

- operating system
- libraries
- device drivers

They are rather seen as common code that can be executed within the context of multiple boxes. Success or failure of the functions called depend on the context, i.e. the current ACLs, as enforced by uVisor.

This group of software items is considered as firmware.

PRIVILEGE LEVEL: Core Firmware



6.4 Security Monitor

This box is in charge of logging and handling security related events:

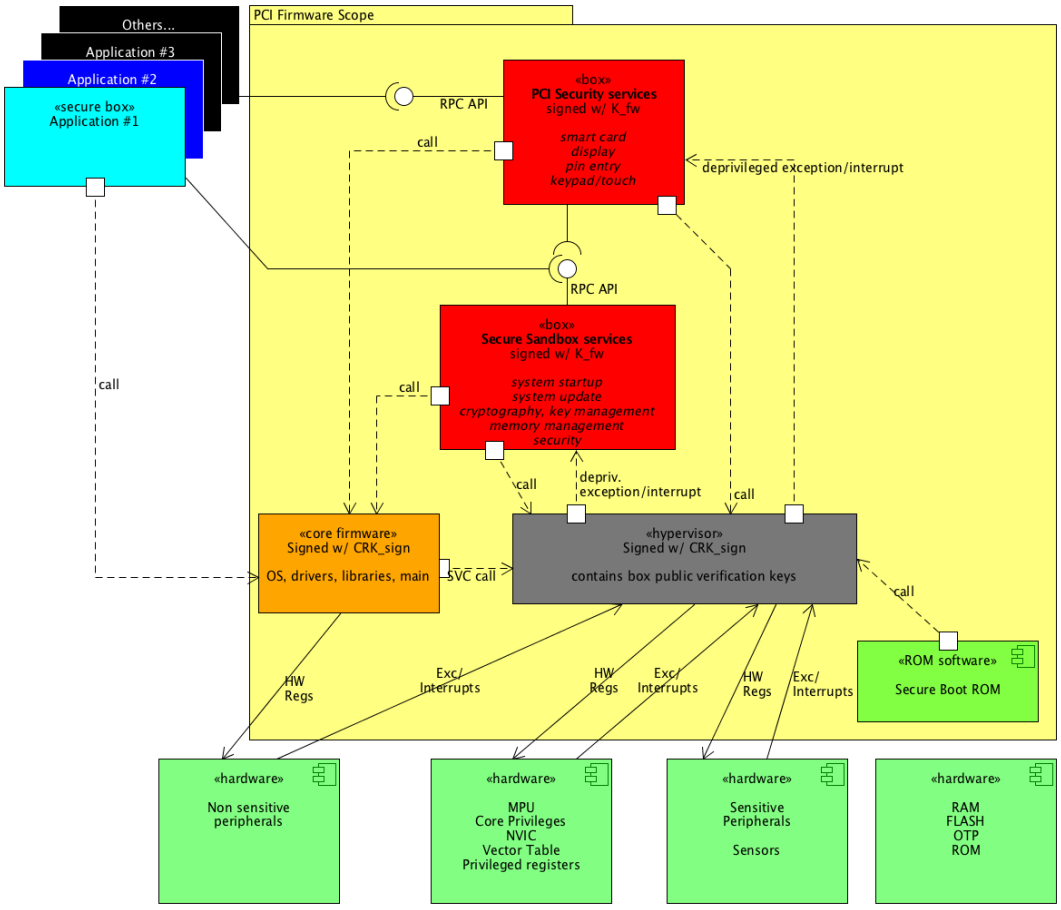
- non maskable interrupts due to security issues
- uvisor related exceptions (access faults)

It allows other boxes to:

- register event handlers to different events
- read the log

This box runs in an independent thread that:

- listens to RPC
- responds to RPC calls and checks the privileges of the caller



6.5 Secure Sandbox services (Generic Security functions)

Modules

- Cryptography
- Global management functions
- I/O API
- Key Manager
- Memory Manager

6.5.1 Detailed Description

The functions available here provide access to sensitive devices and services

- User I/O (touchscreen, keypad, display)
- Cryptography services
- Key manager
- Data storage through memory manager
- Global security management (including firmware update, system startup, response to attacks, integrity checks)
- Register event handlers to different events, and read associated log
  - non maskable interrupts due to security issues
  - uvisor related exceptions (access faults, rpc errors)
- Periodic/one-shot alarm service

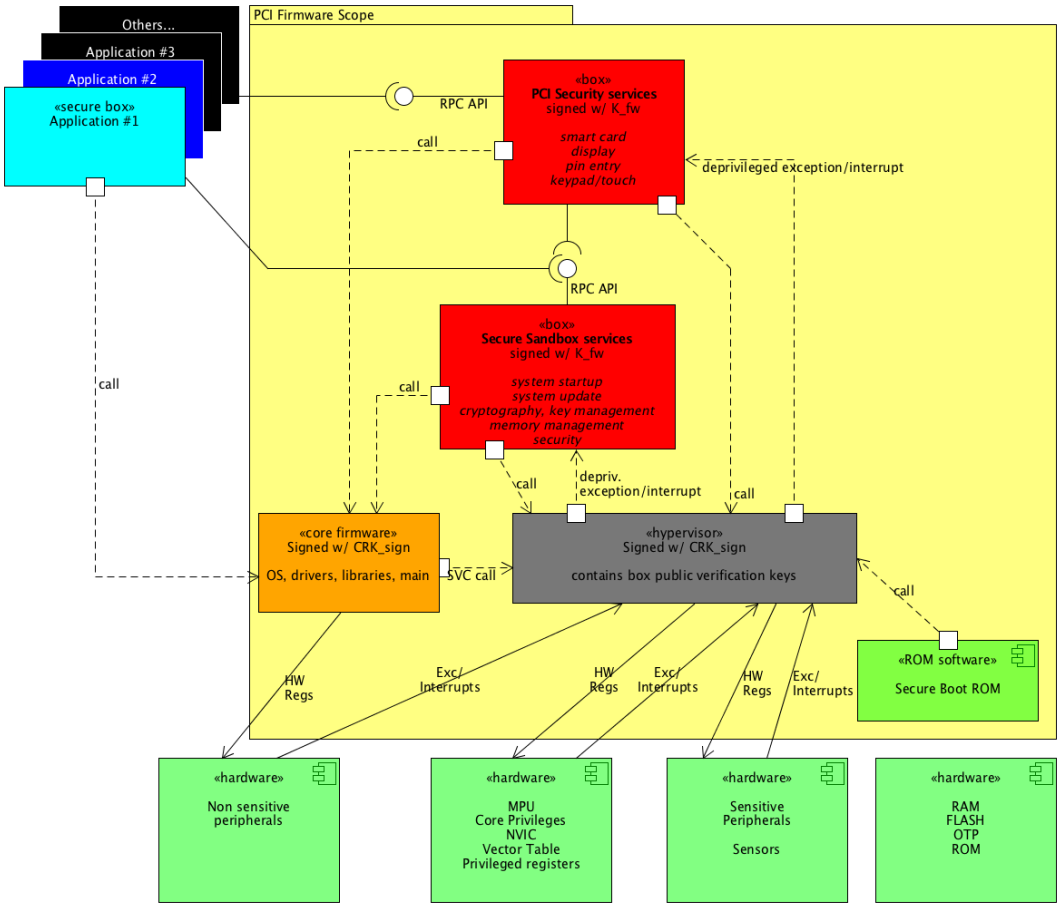
This box is the only box privileged to access some devices. Therefore it proposes various services to interact with those peripherals.

Associated services are accessed through RPC.

This box runs in an independent thread that:

- listens to RPC
- responds to RPC calls and checks the privileges of the caller

PRIVILEGE LEVEL: Box Firmware


Others...
Application #3
Application #2
«secure box»
Application #1
PCI Firmware Scope
RPC API
«box»
PCI Security services
signed w/ K_fw
smart card
display
pin entry
keypad/touch
call
deprivileged exception/interrupt
RPC API
«box»
Secure Sandbox services
signed w/ K_fw
system startup
system update
cryptography, key management
memory management
security
call
call
depriv.
exception/interrupt
call
call
«core firmware»
Signed w/ CRK_sign
OS, drivers, libraries, main
SVC call
«hypervisor»
Signed w/ CRK_sign
contains box public verification keys
call
«ROM software»
Secure Boot ROM
HW
Regs
Exc/
Interrupts
HW
Regs
Exc/
Interrupts
HW
Regs
Exc/
Interrupts
«hardware»
Non sensitive
peripherals
«hardware»
MPU
Core Privileges
NVIC
Vector Table
Privileged registers
«hardware»
Sensitive
Peripherals
Sensors
«hardware»
RAM
FLASH
OTP
ROM

6.5.2 Cryptography

This module offers a generic purpose cryptographic API.

The cryptographic API offers the following services:

- public key based signature/verification using RSA and ECDSA
- symmetric encryption using 3DES and AES
- message digest
- MAC
- DUKPT

It works in coordination with the Key Manager.

6.5.3 Global management functions

List of functions

- int __ssbx_rtc_set_alarm (unsigned int *alarm_id, unsigned int period_minutes, unsigned int type, void(*callback)(void))

  Sets an RTC based alarm.
- int __ssbx_rtc_unset_alarm (unsigned int alarm_id)

  Disables an alarm.
- int ssbx_start (void)

  Starts the system, with various integrity checks.

6.5.3.1 Detailed Description

Additional management services come from the uVisor API: https://github.com/ARMmbed/uvisor/blob/master/docs/api/API.md

6.5.3.2 Function Documentation

6.5.3.2.1 __ssbx_rtc_set_alarm()

```
int __ssbx_rtc_set_alarm (
              unsigned int * alarm_id,
              unsigned int period_minutes,
              unsigned int type,
              void(*)(void) callback )
```

Sets an RTC based alarm.

Parameters

in	*period_minutes*	the number of minutes before the alarm
in	*type*	1 for periodic alarm or 0 for one shot alarm

Returns

ID of the alarm

6.5.3.2.2 __ssbx_rtc_unset_alarm()

```
int __ssbx_rtc_unset_alarm (
              unsigned int alarm_id )
```

Disables an alarm.

Parameters

in	*alarm_id*	The alarm identifier

Returns

-1 if error (caller is not the owner of the alarm), 0 if OK

6.5.3.2.3 ssbx_start()

```
int ssbx_start (
              void  )
```

Starts the system, with various integrity checks.

Returns

See error codes

6.5.4 I/O API

List of functions

- int ssbx_display_display_image (int image_id)

  Displays a pre-defined image on the display.
- int ssbx_display_prompt (unsigned char *kbd_str, int *max_length, int timeout, int message_id, char mask, int font_id)

  Displays a pre-defined prompt on the display and waits for user entry.
- int ssbx_display_write_message (int message_id)

  Displays a pre-defined text message on the display.
- int ssbx_touch_get_entry (char *dest, int *len)

  Gets input from the virtual keypad on the touchscreen.

6.5.4.1 Detailed Description

This module offers some input-output services that allow interaction with users (keypad, touchscreen, display)

6.5.4.2 Function Documentation

6.5.4.2.1 ssbx_display_display_image()

```
int ssbx_display_display_image (
              int image_id )
```

Displays a pre-defined image on the display.

Depending on the current state (see ssbx_set_state), the image will be displayed or an error will be returned.

Parameters

in	*image_id*	The image identifier

Returns

See error codes

6.5.4.2.2 ssbx_display_prompt()

```
int ssbx_display_prompt (
              unsigned char * kbd_str,
              int * max_length,
              int timeout,
              int message_id,
              char mask,
              int font_id )
```

Displays a pre-defined prompt on the display and waits for user entry.

Depending on the current state (see ssbx_set_state), the prompt will be displayed or an error will be returned.

Parameters

	kbd_str	The keyboard string
in	*timeout*	The timeout
in	*message_id*	The message identifier
in	*mask*	The mask character to display in place of user input

Returns

See error codes

6.5.4.2.3 ssbx_display_write_message()

```
int ssbx_display_write_message (
              int message_id )
```

Displays a pre-defined text message on the display.

Depending on the current state (see ssbx_set_state), the text message will be displayed or an error will be returned.

Parameters

in	*message_id*	The message identifier

Returns

See error codes

6.5.4.2.4 ssbx_touch_get_entry()

```
int ssbx_touch_get_entry (
              char * dest,
              int * len )
```

Gets input from the virtual keypad on the touchscreen.

This operation is allowed only if the PCI Secure Services box is not processing a PIN.

Parameters

	dest	The destination
in,out	*len*	The key length

Returns

See error codes

6.5.5 Key Manager

The Key Manager allows secure importation, storage and execution of cryptographic keys in cryptographic protocols offered by the Cryptographic API. It also handles X.509 certificates (importation, verification, public key extraction).

Keys are assigned access rules that define what box can do what with the key (e.g. execute only, import, etc.)

6.5.6 Memory Manager

List of functions

- int ssbx_memsec_alloc (MEMSEC_HANDLE *h, unsigned int size)
  Allocates memory in the NVSRAM.
- int ssbx_memsec_free (MEMSEC_HANDLE *h)
  Releases allocated secure memory (NVSRAM)
- int ssbx_memsec_read (MEMSEC_HANDLE *h, unsigned int offset, unsigned int size)
  Reads from secure memory (NVSRAM)
- int ssbx_memsec_write (MEMSEC_HANDLE *h, unsigned int offset, unsigned int size)
  Writes to secure memory (NVSRAM)

6.5.6.1 Detailed Description

This of module is in charge of securely handling the memory allocation/deallocation in NVSRAM.

6.5.6.2 Function Documentation

6.5.6.2.1 ssbx_memsec_alloc()

```
int ssbx_memsec_alloc (
               MEMSEC_HANDLE * h,
               unsigned int size )
```

Allocates memory in the NVSRAM.

The battery backed NVSRAM holds automatically encrypted/decrypted data. This memory is wiped in case of tamper attack on the system.

This allocator guarantees that data allocated by one box are visible only to this box.

Parameters

in	*h*	pointer to the pointer to the memory handle
in	*size*	size to allocate

Returns

See error codes

6.5.6.2.2 ssbx_memsec_free()

```
int ssbx_memsec_free (
               MEMSEC_HANDLE * h )
```

Releases allocated secure memory (NVSRAM)

Parameters

h	pointer to the memory handle

Returns

See error codes

6.5.6.2.3 ssbx_memsec_read()

```
int ssbx_memsec_read (
                MEMSEC_HANDLE * h,
                unsigned int offset,
                unsigned int size )
```

Reads from secure memory (NVSRAM)

Parameters

	h	pointer to the memory handle
in	*offset*	The offset
in	*size*	size to be read

Returns

See error codes

6.5.6.2.4 ssbx_memsec_write()

```
int ssbx_memsec_write (
                MEMSEC_HANDLE * h,
                unsigned int offset,
                unsigned int size )
```

Writes to secure memory (NVSRAM)

Parameters

	h	pointer to the memory handle
in	*offset*	The offset
in	*size*	The size

Returns

See error codes

6.6 PCI Security Services, Security functions dedicated to PCI PTS POI security

Modules

- EMV-Level 1 Smart Card
- Magnetic Stripe
- PIN handling

6.6.1 Detailed Description

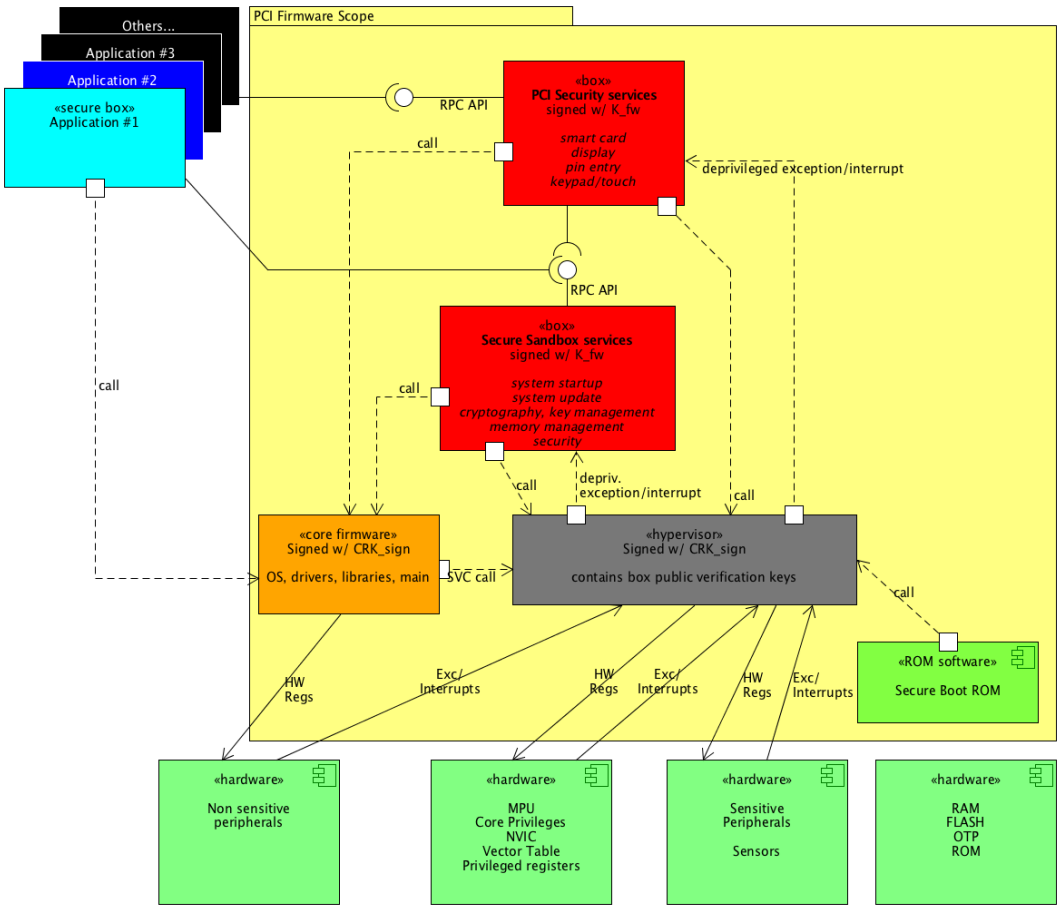
Security functions dedicated to PCI PTS POI security are provided in this module, in particular for the PIN and Magnetic Stripe data handling, the Smart Card communication.

This box is the only box privileged to access some devices. Therefore it proposes various services to interact with those peripherals.

Associated services are accessed through RPC.

This box runs in an independent thread that:

- listens to RPC
- responds to RPC calls and checks the privileges of the caller



6.6.2 EMV-Level 1 Smart Card

List of functions

- int pci_smartcard_config (SCARD_CONF sconf)

  Configures the Smart Card communication.
- int pci_smartcard_transact_APDU (unsigned char APDU_Res[ ], unsigned int ∗APDU_Res_len, unsigned int timeout, unsigned char APDU_Req[ ], unsigned int ∗APDU_Req_len)

  Performs an APDU exchange with the smart card. This function conforms to the EMV-Level 1 specification.
- int pci_smartcard_wait_card_insertion (unsigned int timeout)

  Wait for the smart card being inserted.
- int pci_smartcard_wait_card_removal (unsigned int timeout)

  Wait for the smart card being removed.

6.6.2.1 Detailed Description

This of module is in charge of handling communication with Smart Cards while conforming to the EMV Level-1 specification.

6.6.2.2 Function Documentation

6.6.2.2.1 pci_smartcard_config()

```
int pci_smartcard_config (
              SCARD_CONF sconf )
```

Configures the Smart Card communication.

Parameters

in	*sconf*	The smart card configuration

Returns

See error codes

6.6.2.2.2 pci_smartcard_transact_APDU()

```
int pci_smartcard_transact_APDU (
              unsigned char APDU_Res[],
              unsigned int * APDU_Res_len,
              unsigned int timeout,
              unsigned char APDU_Req[],
              unsigned int * APDU_Req_len )
```

Performs an APDU exchange with the smart card. This function conforms to the EMV-Level 1 specification.

Parameters

	APDU_Res	The apdu resource

Parameters

	APDU_Res_len	The apdu resource length
in	*timeout*	The timeout
	APDU_Req	The apdu request
in	*APDU_Req_len*	The apdu request length

Returns

See error codes

6.6.2.2.3 pci_smartcard_wait_card_insertion()

```
int pci_smartcard_wait_card_insertion (
              unsigned int timeout )
```

Wait for the smart card being inserted.

Parameters

in	*timeout*	The timeout

Returns

See error codes

6.6.2.2.4 pci_smartcard_wait_card_removal()

```
int pci_smartcard_wait_card_removal (
              unsigned int timeout )
```

Wait for the smart card being removed.

Parameters

in	*timeout*	The timeout

Returns

See error codes

6.6.3 PIN handling

List of functions

- int pci_authenticate_issuer_and_icc_public_key (unsigned char *issuer_certificate, int issuer_certificate_len, unsigned char *issuer_remainder, int issuer_remainder_len, unsigned char *issuer_exponent, int issuer_exponent_len, unsigned char *icc_certificate, int icc_certificate_len, unsigned char *icc_remainder, int icc_remainder_len, unsigned char *icc_exponent, int icc_exponent_len, CertificationAuthorities *CA_key)

  Authenticates both issuer public key and ICC public key.
- int pci_get_online_pin (unsigned char pan[ ], int pan_len, unsigned char encrypted_pin[ ], int *encrypted_pin_len)

  Gets the "online" PIN, i.e. encrypted with the DUKPT algorithm. The resulting encrypted PIN block is encoded into ISO 9564 Format 0.
- int pci_pin_entry (int timeout_entry, int timeout_pin, int entry_device, int display_device)

  Displays a message to the user instructing to enter the PIN. Then gets the pin from the specified entry device.
- int pci_verify_offline_pin (unsigned char challenge[ ], int challenge_len, int encrypt_mode, unsigned int timeout)

  Verifies user PIN using the Smart Card.

6.6.3.1 Detailed Description

This module is in charge of handling PIN entry and processing according to the applicable PCT-PTS-POI standard.

6.6.3.2 Function Documentation

6.6.3.2.1 pci_authenticate_issuer_and_icc_public_key()

```
int pci_authenticate_issuer_and_icc_public_key (
                unsigned char * issuer_certificate,
                int issuer_certificate_len,
                unsigned char * issuer_remainder,
                int issuer_remainder_len,
                unsigned char * issuer_exponent,
                int issuer_exponent_len,
                unsigned char * icc_certificate,
                int icc_certificate_len,
                unsigned char * icc_remainder,
                int icc_remainder_len,
                unsigned char * icc_exponent,
                int icc_exponent_len,
                CertificationAuthorities * CA_key )
```

Authenticates both issuer public key and ICC public key.

This function authenticates both issuer public key and ICC public key by a hardcoded Certification Authority Public key.

As per EMV Book2, ICC can use either ICC Public Key or ICC PIN Encipherment Key for Offline Pin Authentication.

EMV Application kernel Identifies which ICC Public key to be used for Offline PIN Encipherment. But, SHIDDaemon uses ICC key only after Successful Authentication. Since CA key is hard coded inside SHIDDaemon, Security is not compromised even when Application kernel provides part of keys and certificate as input to SHIDDaemon.

After use, the PIN buffered gets zeroed using memset.

Parameters

in	*issuer_certificate*	Issuer Public key certificate
in	*issuer_certificate_len*	Issuer Public key certificate length
in	*icc_certificate*	ICC Public key certificate
in	*icc_certificate_len*	ICC Public key certificate length

Returns

Error code

6.6.3.2.2 pci_get_online_pin()

```
int pci_get_online_pin (
                unsigned char pan[],
                int pan_len,
                unsigned char encrypted_pin[],
                int * encrypted_pin_len )
```

Gets the ”online” PIN, i.e. encrypted with the DUKPT algorithm. The resulting encrypted PIN block is encoded into ISO 9564 Format 0.

It uses the PIN buffered using pci_pin_entry.

The context is checked so that only the right caller that has started the PIN entry process can call this function.

After use, the PIN buffered gets zeroed using memset, as well as DUKPT temporary buffers

Parameters

in	*store_id*	The DUKPT store identifier
	PIN	The clear-text PIN
	PAN	The PAN
in	*PAN_len*	The PAN length

Returns

See error codes

6.6.3.2.3 pci_pin_entry()

```
int pci_pin_entry (
                int timeout_entry,
                int timeout_pin,
                int entry_device,
                int display_device )
```

Displays a message to the user instructing to enter the PIN. Then gets the pin from the specified entry device.

The PIN must be entered before a certain amount of time. If not, the function returns.

In case of error, no PIN data remains anywhere in memory. In case of success, the PIN data is kept during at most timeout_pin

Parameters

in	*timeout_entry*	The timeout for the entry of the PIN by the user
in	*timeout_pin*	The maximum time during which the PIN is kept in memory. Otherwise it gets zeroed using memset.
in	*entry_device*	The entry device
in	*display_device*	The display device

Returns

See error codes

6.6.3.2.4 pci_verify_offline_pin()

```
int pci_verify_offline_pin (
              unsigned char challenge[],
              int challenge_len,
              int encrypt_mode,
              unsigned int timeout )
```

Verifies user PIN using the Smart Card.

This function verifies encrypted pin block with ICC. This function receives Verify APDU header and embedded encrypted pin block before sending it to ICC.

Parameters

out	*Verify_APDU_Res*	APDU response
	Verify_APDU_Res_len	The verify apdu resource length
in	*timeout*	Maximum time allowed for APDU transaction. This value should be in Seconds.
	Verify_APDU_Req	The verify apdu request
out	*Verify_APDU_len*	APDU response length

Returns

Error code

6.6.4 Magnetic Stripe

PLANNED IN V2: This module is in charge of handling the magnetic stripe reading conforming to the SRED requirements.

6.7 uVisor API

Data Structures

- struct UvisorBoxAclItem

Macros

- #define UVISOR_BOX_CONFIG(box_nameconst UvBoxAclItem module_acl_listuint32_t module_stack_↩ size, struct __your_context, verif_key_id)

  Secure box configuration.
- #define UVISOR_BOX_NAMESPACE(static const char const namespace)

  Specify the namespace for a box. C/C++ pre-processor macro (pseudo-function)
- #define UVISOR_SET_MODE(uvisor_mode)

  Set mode for the uVisor [temporary].
- #define UVISOR_SET_MODE_ACL(uvisor_mode, const UvBoxAcl main_box_acl_list)

  Set mode for the uVisor and provide background ACLs for the main box.

List of types

- typedef uint32_t UvisorBoxAcl

List of functions

- int check_acl (uint32_t *p_acl, uint32_t keyindex)

  Hook called by uVisor during loading of secure boxes.
- int rpc_fncall_waitfor (const TFN_Ptr fn_ptr_array[ ], size_t fn_count, int *box_id_caller, uint32_t timeout_ms)

  Handle incoming RPC, setting the parameter box_id_caller to the caller box ID.
- int uvisor_box_id_self (void)

  Get the ID of the current box.
- int uvisor_box_namespace (int box_id, char *box_namespace, size_t length)

  Copy the namespace of the specified box to the provided buffer.
- int uvisor_box_signingkey (int box_id, int *keyindex)

  Get the signing key ID of the specified box, hence its box privilege.
- void vIRQ_ClearPendingIRQ (uint32_t irqn)

  Clear pending status of IRQn.
- void vIRQ_DisableIRQ (uint32_t irqn)

  Disable IRQn for the currently active box.
- void vIRQ_EnableIRQ (uint32_t irqn)

  Enable IRQn for the currently active box.
- int vIRQ_GetLevel (void)

  Get level of currently active IRQn, if any.
- uint32_t vIRQ_GetPendingIRQ (uint32_t irqn)

  Get pending status of IRQn.
- uint32_t vIRQ_GetPriority (uint32_t irqn)

  Get priority level of IRQn.
- uint32_t vIRQ_GetVector (uint32_t irqn)

  Get the ISR registered for IRQn.
- void vIRQ_SetPendingIRQ (uint32_t irqn)

  Set pending status of IRQn.
- void vIRQ_SetPriority (uint32_t irqn, uint32_t priority)

  Set priority level of IRQn.
- void vIRQ_SetVector (uint32_t irqn, uint32_t vector)

  Register an ISR to the currently active box.

6.7.1 Detailed Description

The payment application is to be developed by the payment terminal vendor. It dialogs with the 2 secure boxes through remote procedure calls (RPC). The API is described below in this document. Shortly this API allows to display trusted messages on the display, perform EMV-level 1 compliant APDU exchanges with the smart card, perform PIN processing, capture Mag Stripe... basically everything required to perform an EMV Level-2 compliant payment.

Here you can find detailed documentation for:

1. `Configuration macros`, to configure a secure box and protect data and peripherals.
2. `Box Identity`, to retrieve a box-specific ID or the namespace of the current or calling box.
   - A box identity identifies a security domain uniquely and globally.
   - The box identity API can be used to determine the source box of an inbound secure gateway call. This can be useful for implementing complex authorization logic between mutually distrustful security domains.
   - uVisor provides the ability to retrieve the box ID of the current box (`uvisor_box_id_self`), or of the box that called the current box through an RPC gateway via the `box_id_caller` parameter of `rpc_fncall_waitfor`.
   - The box ID number is not constant and can change between reboots. But, the box ID number can be used as a token to retrieve a constant string identifier, known as the box namespace.
   - A box namespace is a static, box-specific string, that can help identify which box has which ID at run-time. In the future, the box namespace will be guaranteed to be globally unique.
   - A full example using this API is available at `mbed-os-example-uvisor-number-store`.
3. `Low level APIs`, to access uVisor functions that are not available to unprivileged code (interrupts, restricted system registers).
4. `Type definitions`.
5. `Error codes`.

Error reason	Error code
PERMISSION_DENIED	1
SANITY_CHECK_FAILED	2
NOT_IMPLEMENTED	3
NOT_ALLOWED	4
FAULT_MEMMANAGE	5
FAULT_BUS	6
FAULT_USAGE	7
FAULT_HARD	8
FAULT_DEBUG	9

6.7.2 Data Structure Documentation

6.7.2.1 struct UvisorBoxAclItem

{ item_description }

Data Fields

UvisorBoxAcl	acl	
uint32_t	length	
const volatile void *	start	

6.7.3 Macro Definition Documentation

6.7.3.1 UVISOR_BOX_CONFIG

```
#define UVISOR_BOX_CONFIG(
                box_nameconst UvBoxAclItem module_acl_listuint32_tmodule_stack_size,
                struct __your_context,
                verif_key_id )
```

Secure box configuration.

C/C++ pre-processor macro (pseudo-function)

Parameters

box_name	Secure box name
module_acl_list	List of ACLs for the module
module_stack_size	Required stack size for the secure box
__your_context	[optional] Type definition of the struct hosting the box context data
verif_key_id	the Id of the key to use for the verification of the signature of the box #define KEYINDEX_FW 0x4E5F739aUL #define KEYINDEX_TRUSTEDAPP 0xebf78ac4UL #define KEYINDEX_OTHERAPP 0x9cfb3459UL

Example: "' #include "uvisor-lib/uvisor-lib.h"

// Required stack size #define BOX_STACK_SIZE 0x100

// Define the box context. typedef struct { uint8_t secret[SECRET_SIZE]; bool initialized; State_t current_state } BoxContext;

// Create the ACL list for the module. static const UvBoxAclItem g_box_acl[] = { {PORTB, sizeof(*PORTB), UVISOR_TACLDEF_PERIPH}, {RTC, sizeof(*RTC), * UVISOR_TACLDEF_PERIPH}, {LPTMR0, sizeof(*↩ LPTMR0), UVISOR_TACLDEF_PERIPH}, };

// Configure the secure box compartment. UVISOR_BOX_NAMESPACE("com.example.my-box-name"); UVI↩ SOR_BOX_CONFIG(my_box_name, g_box_acl, BOX_STACK_SIZE, BoxContext);

"'

6.7.3.2 UVISOR_BOX_NAMESPACE

```
#define UVISOR_BOX_NAMESPACE(
                static const char constnamespace )
```

Specify the namespace for a box. C/C++ pre-processor macro (pseudo-function)

Parameters

namespace	The namespace of the box

The namespace of the box must be a null-terminated string no longer than MAX_BOX_NAMESPACE_LENGTH (including the terminating null).

The namespace must also be stored in public flash. uVisor will verify that the namespace is null-terminated and stored in public flash at boot-time, and will halt if the namespace fails this verification. For now, use a reverse domain name for the box namespace.

If you don't have a reverse domain name, use a GUID string identifier. We currently don't verify that the namespace is globally unique, but we will perform this validation in the future.

Use of this configuration macro before UVISOR_BOX_CONFIG is required. If you do not wish to give your box a namespace, specify NULL as the namespace to create an anonymous box.

Example:

```
#include "uvisor-lib/uvisor-lib.h"

// Configure the secure box.
UVISOR_BOX_NAMESPACE("com.example.my-box-name");
UVISOR_BOX_CONFIG(my_box_name, UVISOR_BOX_STACK_SIZE);
```

6.7.3.3 UVISOR_SET_MODE

```
#define UVISOR_SET_MODE(
                uvisor_mode )
```

Set mode for the uVisor [temporary].

C/C++ pre-processor macro (object declaration)

Parameters

in	*uvisor_mode*	The uvisor mode: • UVISOR_DISABLED = disabled [default] • UVISOR_PERMISSIVE = permissive [currently n.a.] • UVISOR_ENABLED = enabled

Example:

```
#include "uvisor-lib/uvisor-lib.h"

// Set the uVisor mode.
UVISOR_SET_MODE(UVISOR_ENABLED);
```

Note:

1. This macro is only needed temporarily (uVisor disabled by default) and will be removed in the future.
2. This macro must be used only once in the top level yotta executable.

6.7.3.4 UVISOR_SET_MODE_ACL

```
#define UVISOR_SET_MODE_ACL(
                uvisor_mode,
                const UvBoxAclmain_box_acl_list )
```

Set mode for the uVisor and provide background ACLs for the main box.

C/C++ pre-processor macro (object declaration)

Parameters

in	*uvisor_mode*	The uvisor mode • UVISOR_DISABLED< = disabled [default] • UVISOR_PERMISSIVE = permissive [currently n.a.] • UVISOR_ENABLED = enabled
	main_box_acl_list	List of ACLs for the main box (background ACLs)

Example:

```
#include "uvisor-lib/uvisor-lib.h"
// Create background ACLs for the main box.
static const UvBoxAclItem g_background_acl[] = {
  {UART0,       sizeof(*UART0), UVISOR_TACL_PERIPHERAL},
  {UART1,       sizeof(*UART1), UVISOR_TACL_PERIPHERAL},
  {PIT,         sizeof(*PIT),   UVISOR_TACL_PERIPHERAL},
};

// Set the uVisor mode.
UVISOR_SET_MODE_ACL(UVISOR_ENABLED, g_background_acl);
```

Note:

1. This macro is only needed temporarily (uVisor disabled by default) and will be removed in the future.
2. This macro must be used only once in the top level yotta executable.

6.7.4 Typedef Documentation

6.7.4.1 UvisorBoxAcl

```
typedef uint32_t UvisorBoxAcl
```

{ item_description }

6.7.5 Function Documentation

6.7.5.1 check_acl()

```
int check_acl (
                uint32_t * p_acl,
                uint32_t keyindex )
```

Hook called by uVisor during loading of secure boxes.

It is the responsibility of the platform developer to implement this function to restrict ACLs based on the box privilege of the box being loaded.

Parameters

	p_acl	Pointer to the ACL of the box being loaded
in	*keyindex*	The key used to verify the box (hence its box privilege)

Returns

0 if the ACL conforms to the ACL policy

6.7.5.2 rpc_fncall_waitfor()

```
int rpc_fncall_waitfor (
                const TFN_Ptr fn_ptr_array[],
                size_t fn_count,
                int * box_id_caller,
                uint32_t timeout_ms )
```

Handle incoming RPC, setting the parameter box_id_caller to the caller box ID.

When deciding which memory to provide for rpc_fncall_waitfor to use when writing `box_id_caller`, strongly prefer thread local storage when multiple threads in a box can handle incoming RPC.

Parameters

in	*fn_ptr_array*	The function pointer array
in	*fn_count*	The function count
	box_id_caller	The box identifier of the caller. After a call, box_id_caller is set to the box ID of the calling box (the source box of the RPC). This is set before the RPC is dispatched, so that the RPC target function can read from this location to determine the calling box ID. This parameter is optional.
in	*timeout_ms*	The timeout milliseconds

Returns

{ description_of_the_return_value }

6.7.5.3 uvisor_box_id_self()

```
int uvisor_box_id_self (
                void  )
```

Get the ID of the current box.

Returns

The ID of the current box

6.7.5.4 uvisor_box_namespace()

```
int uvisor_box_namespace (
                int box_id,
                char * box_namespace,
                size_t length )
```

Copy the namespace of the specified box to the provided buffer.

Parameters

in	*box_id*	The ID of the box you want to retrieve the namespace of
	box_namespace	The buffer where the box namespace will be copied to
in	*length*	The length in bytes of the provided box_namespace buffer

Returns

Return how many bytes were copied into box_namespace.

Return values

UVISOR_ERROR_INVALID_BOX_ID	if the provided box ID is invalid.
UVISOR_ERROR_BUFFER_TOO_SMALL	if the provided box_namespace is too small to hold MAX_BOX_NAMESPACE_LENGTH bytes.
UVISOR_ERROR_BOX_NAMESPACE_ANONYMOUS	if the box is anonymous.

6.7.5.5 uvisor_box_signingkey()

```
int uvisor_box_signingkey (
                int box_id,
                int * keyindex )
```

Get the signing key ID of the specified box, hence its box privilege.

Parameters

in	*box_id*	The ID of the box you want to retrieve the signing key ID.
	keyindex	Location where to receive the requested key ID

Returns

Status of execution

Return values

UVISOR_ERROR_INVALID_BOX_ID	if the provided box ID is invalid.

6.7.5.6 vIRQ_ClearPendingIRQ()

```
void vIRQ_ClearPendingIRQ (
              uint32_t irqn )
```

Clear pending status of IRQn.

Parameters

in	*irqn*	IRQn

6.7.5.7 vIRQ_DisableIRQ()

```
void vIRQ_DisableIRQ (
              uint32_t irqn )
```

Disable IRQn for the currently active box.

Parameters

in	*irqn*	IRQn

6.7.5.8 vIRQ_EnableIRQ()

```
void vIRQ_EnableIRQ (
              uint32_t irqn )
```

Enable IRQn for the currently active box.

Parameters

in	*irqn*	IRQn

6.7.5.9 vIRQ_GetLevel()

```
int vIRQ_GetLevel (
              void  )
```

Get level of currently active IRQn, if any.

Returns

The priority level of the currently active IRQn, if any; -1 otherwise

6.7.5.10 vIRQ_GetPendingIRQ()

```
uint32_t vIRQ_GetPendingIRQ (
              uint32_t irqn )
```

Get pending status of IRQn.

Parameters

in	*irqn*	IRQn

Returns

pending status of IRQn

6.7.5.11 vIRQ_GetPriority()

```
uint32_t vIRQ_GetPriority (
              uint32_t irqn )
```

Get priority level of IRQn.

Parameters

in	*irqn*	IRQn

Returns

The priority level of IRQn, if available; 0 otherwise

6.7.5.12 vIRQ_GetVector()

```
uint32_t vIRQ_GetVector (
              uint32_t irqn )
```

Get the ISR registered for IRQn.

Parameters

in	*irqn*	IRQn

Returns

The ISR registered for IRQn, if present; 0 otherwise

6.7.5.13 vIRQ_SetPendingIRQ()

```
void vIRQ_SetPendingIRQ (
              uint32_t irqn )
```

Set pending status of IRQn.

Parameters

in	*irqn*	IRQn

6.7.5.14 vIRQ_SetPriority()

```
void vIRQ_SetPriority (
              uint32_t irqn,
              uint32_t priority )
```

Set priority level of IRQn.

Parameters

in	*irqn*	IRQn
in	*priority*	Priority level (minimum: 1)

6.7.5.15 vIRQ_SetVector()

```
void vIRQ_SetVector (
              uint32_t irqn,
              uint32_t vector )
```

Register an ISR to the currently active box.

Parameters

in	*irqn*	IRQn
in	*vector*	Interrupt handler; if 0 the IRQn slot is de-registered for the current box

7. PCI PTS POI 5.0 Guidance (DRAFT, to be modified)

This section explains how Deeptrust covers the PCI PTS POI 5.0 compliance, requirement by requirement.

8. References

[DOC1] uVisor Current Release as of Nov 29th https://github.com/ARMmbed/uvisor/blob/master/docs/api/API.md 11-29-2016

[2] ISO 9564-1:2011 https://en.wikipedia.org/wiki/ISO_9564 2011

[3] PIN Transaction Security (PTS) Point of Interaction (POI) Version 5 https://www.pcisecuritystandards.org/documents/PCI_PTS_POI_SRs_v5.pdf?agreement=true&time=1480425082529 09-2016

[4] PCI Linux User Guide Revision F Maxim Integrated Products UG21T20 01-24-2015

[5] MAX32550 User Guide Revision E Maxim Integrated Products UG25H05 07-01-2016

[6] ANSI X9.24-1:2009. http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.24-1%3A2009 2009

[7] Cortex™-M3 Devices Generic User Guide http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337h/index.html 2010

[8] Cortex™-M4 Devices Generic User Guide http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/index.html 2011

[9] Book 1 Application Independent ICC to Terminal Interface Requirements - November 2011

[10] uVisor API https://github.com/ARMmbed/uvisor/blob/master/docs/api/API.md

[11] MAX32550 Secure ROM User Guide Revision H Maxim Integrated Products UG25H04 10-25-2013

[12] MAX32550 security evaluation report Revision E Maxim Integrated Products RP25T05 05-04-2015

Index