1 - TrustZone on ARMv8 Cortex-M (Cortex M33)

The TrustZone for Cortex-M is a security extension for Armv8-M architecture that is optimized for ultra-low power embedded applications. It enables software security domains that restrict access to secure memory and I/O only for trusted software.

Arm TrustZone technology for Armv8-M divide hardware into two sections: the secure and the non-secure. The division between the two sections is memory-mapped, so there will be addresses associated with both the secure and non-secure parts. Importantly, all resources of a microcontroller, including Flash memory, SRAM, peripherals, interrupts, and others, are allocated either to the secure or the non-secure section. The assignment of a resource to one of the two sections can be determined by a combination of hardware and software and affects how that resource accesses other resources: a non-secure resource can only access other non-secure resources, while a secure resource can access all resources.

TrustZone for Armv8-M:

- preserves low interrupt latencies for both Secure and Non-secure domains.
- does not impose code overhead, cycle overhead or the complexity of a virtualization based solution.
- introduces the Secure Gateway (SG) processor instruction for calls to the secure domain.

1.1 - The Secure and Non-Secure World

In a simplified view, the executed code address determines the security state of the CPU, that is either secure or non-secure:

- If the CPU runs code in a non-secure memory, the CPU is in non-secure state.
- If the CPU runs code in a secure memory, the CPU is in secure state.

The Armv8-M technology defines the following address security attributes:

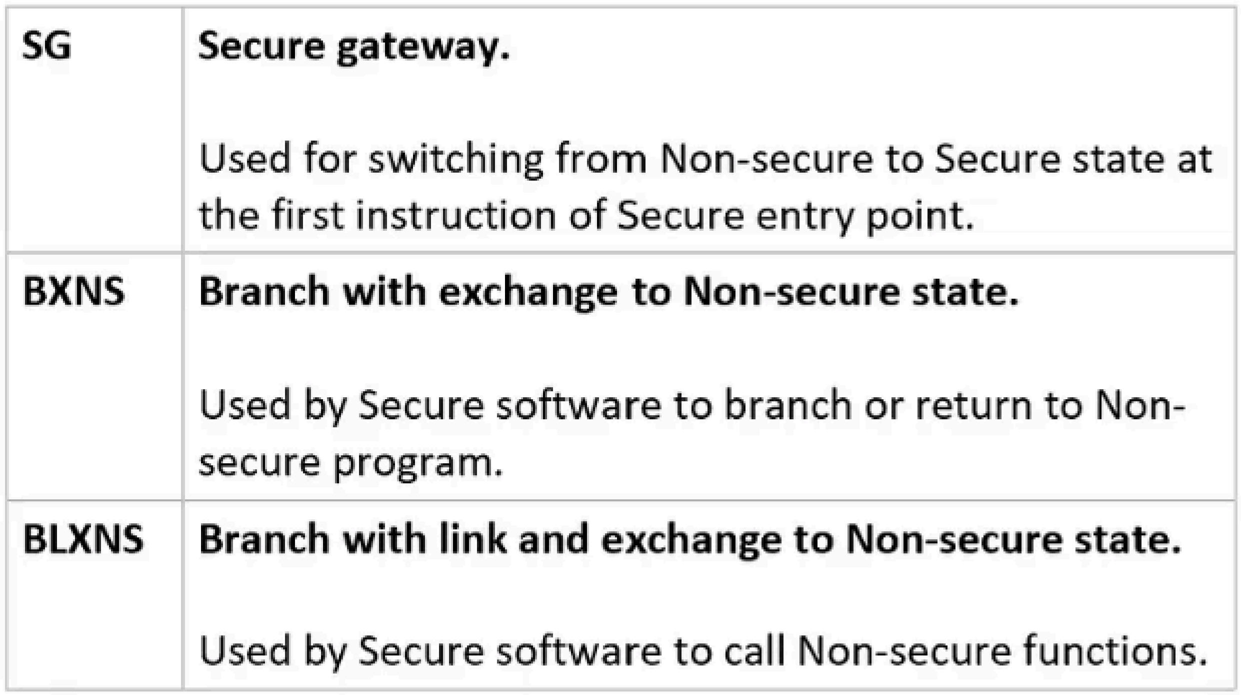
- Secure (S) addresses are used for memory and peripherals that are only accessible by secure code or secure masters. Secure transactions are those that originate from masters operating as secure.
- Non-secure callable (NSC) NSC is a special type of secure location. This type of memory is the only type for which an Armv8-M processor permits to hold an SG (secure gateway) instruction that enables software to transition from non-secure to secure state. This SG

instruction can be used to prevent non-secure applications from branching into invalid entry points. When a non-secure code calls a function in the secure side:
  - The first instruction in the API must be an SG instruction.
  - The SG instruction must be in an NSC region. Secure code also provides non-secure callable functions to provide secure service accesses to non-secure code.
- Non-secure (NS) addresses are used for memories and peripherals accessible by all software running on the device. Non-secure transactions originate from masters operating as non-secure or from secure masters accessing a non-secure address (data transaction only, not fetch instructions). Non-secure transactions are only permitted to access non-secure addresses. Non-secure transactions cannot access to secure addresses.

1.1.1 - New instructions

There are two types of new instuctions, one regards the switching between the Non-secure world and Secure world (and viceversa) and the other regards the testing of the permisison of memory locations in terms of security.

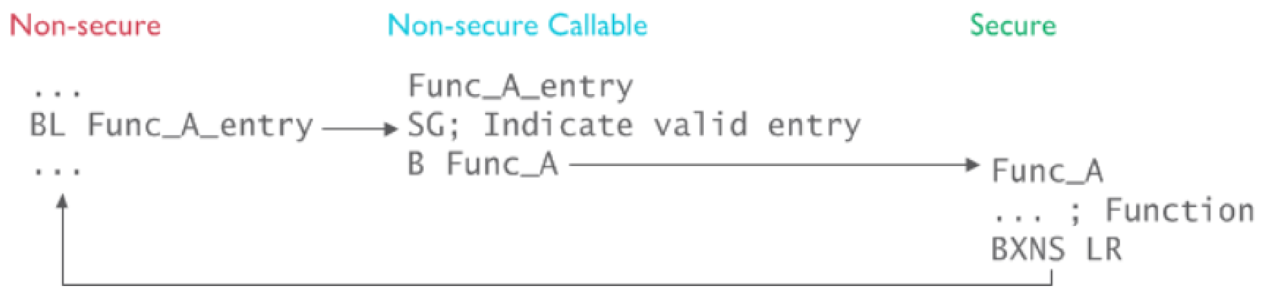
First are presented the switching related ones.



SG	**Secure gateway.** Used for switching from Non-secure to Secure state at the first instruction of Secure entry point.
BXNS	**Branch with exchange to Non-secure state.** Used by Secure software to branch or return to Non-secure program.
BLXNS	**Branch with link and exchange to Non-secure state.** Used by Secure software to call Non-secure functions.

- SG: Secure Gateway marks a valid branch target for branches from Non-secure code that wants to call Secure code.
- BXNS and BLXNS:Syntax: BXNS <Rm> BLXNS <Rm>
  Where: Rm is a register containing an address to branch to.
  The BLXNS instruction calls a subroutine at an address contained in Rm and conditionally causes a transition from the Secure to the Non-secure state.

For both BXNS and BLXNS, Rm[0] indicates a transition to Non-secure state if value is 0, otherwise the target state remains Secure. If transitioning to Non-secure, BLXNS pushes the return address and partial PSR to the Secure stack. These instructions are available for Secure state only. When the processor is in Non-secure state, these instructions are UNDEFINED and triggers a UsageFault if executed.



In figure is shown the usual execution flow.

The other instructions are the Test Target family, instructions that are used for testing the security attributes of an address.

The intructions are TT, TTT, TTA, and TTAT all with the same syntax {op}{cond} Rd, Rn. Rd is the destination register into which the status result of the target test is written and Rn is the base register. These instructions are implemented with some a hardware support (SAU) that is explained below.

- TT Test Target (TT) queries the Security state and access permissions of a memory location.
- TTT Test Target Unprivileged (TTT) queries the Security state and access permissions of a memory location for an unprivileged access to that location.
- TTA In an implementation with the Security Extension, Test Target Alternate Domain (TTA) queries the Security state and access permissions of a memory location for a Non-secure access to that location. These instructions are only valid when executing in Secure state, and are UNDEFINED if used from Non-secure state.
- TTAT In an implementation with the Security Extension, Test Target Alternate Domain Unprivileged (TTAT) queries the Security state and access permissions of a memory location for a Non-secure and unprivileged access to that location. These instructions are only valid when executing in Secure state, and are UNDEFINED if used from Non-secure state.

1.1.2 - The new stack frame

The Arm TrustZone allows user to divide memory map into Secure and Non-Secure regions and Secure and Non-Secure code can be executed so the necessity to create two stacks arises. Stack management expands from two stack pointers in original Cortex-M (Main Stack Pointer

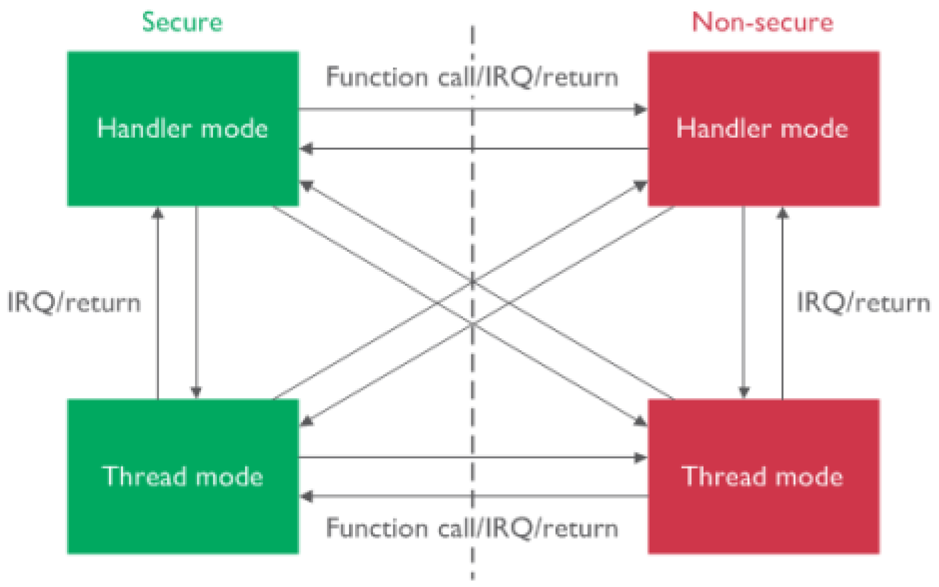
(MSP) and Process Stack Pointer (PSP)) to four, providing the above pair individually to both Secure and Non-Secure. These instances are switched automatically by the core during transitioning from Non-Secure to Secure and back.
As part of ARM TrustZone technology for ARMv8-M, there is also a stack limit checking feature.

1.1.3 - SecureFault and secure interrupts

The introduction of TrustZone is also reflected in the implementation of faults and interrupts. A new exception is introducede, the SecureFault. This exception is triggered by the various security checks that are performed. It is triggered, for example, when jumping from Non-secure code to an address in Secure code that is not marked as a valid entry point. Most systems choose to treat a SecureFault as a terminal condition that either halts or restarts the system. Any other handling of the SecureFault must be checked carefully to make sure that it does not inadvertently introduce a security vulnerability. SecureFaults always target the Secure state.

The Armv8-M Security Extension adds security through code and data protection features. A processor with the Security Extension supports both Non-secure and Secure states, which are orthogonal to the traditional thread and handler modes. The four modes of operation are:

- Non-secure Thread mode.
- Non-secure Handler mode.
- Secure Thread mode.
- Secure Handler mode.



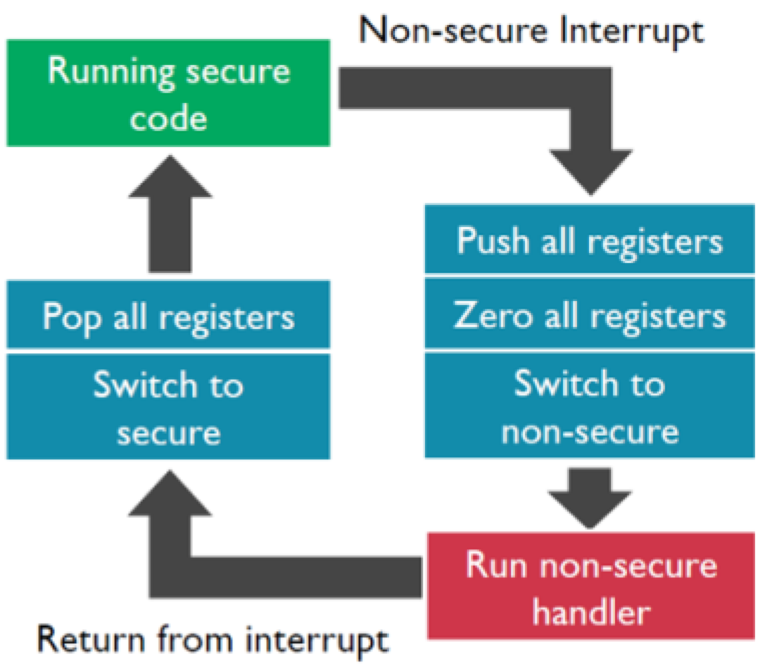
This means that for example during the handling of an interrupt in the secure state, the ISR can call a non-secure function.

When the Security Extension is implemented, the following happens:

- The processor resets into Secure state.
- Some registers are banked between Security states. There are two separate instances of the same register, one in Secure state and one in Non-secure state.
- The architecture allows the Secure state to access the Non-secure versions of banked registers.
- Interrupts can be configured to target one of the two Security states.
- Some faults are banked between Security states or are configurable.
- Secure memory can only be accessed from Secure state.

More in detail the Nested Vectored Interrupt Controller (NVIC) is also extended for security as state transitions can also happen due to exceptions and interrupts.

- Each interrupt can be configured as Secure or Non-secure, and is determined by the Interrupt Target Non-secure (NVIC_ITNS) register, which is only programmable in the Secure world. There are no restrictions regarding whether a Non-secure or Secure interrupt can take place when the processor is running Non-secure or Secure code.
- If the arriving exception or interrupt has the same state as the current processor state, then the exception sequence is similar to the previous M-series processors.
- The main difference occurs when a non-secure interrupt takes place and is handled by the processor during the execution of secure code. In this case, the processor automatically pushes all secure information onto the secure stack and erases the contents from the register banks — this mechanism avoids any leakage of information.

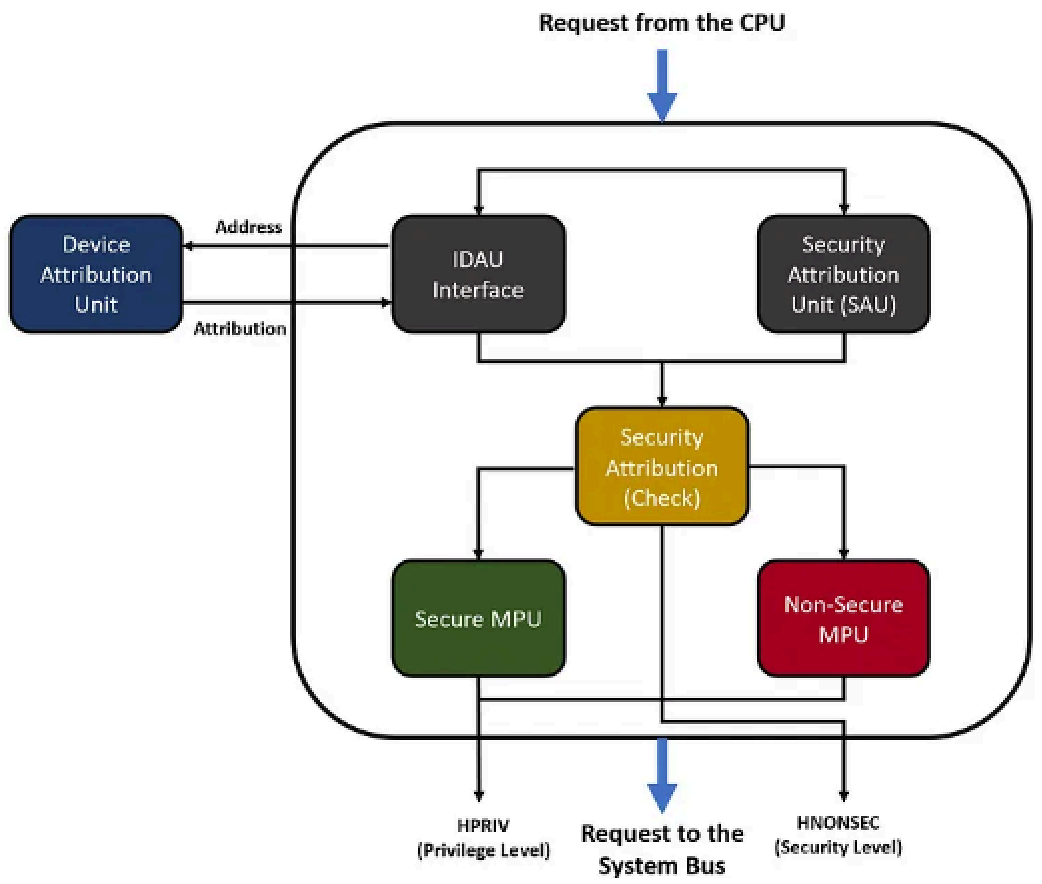


1.2 The SAU

Configuration of memory security attributes is done via two hardware blocks called "security attribution unit" (SAU) and/or "implementation defined attribution unit" (IDAU). The Secure Attribution Unit (SAU) is a programmable unit integrated in the processor core used to define the security status of up to eight memory regions. These regions can be configured to be Secure, Non-Secure or Non-Secure-Callable. Without the SAU it would not be possible to implement TrustZone.
After assigning ‘security attributes’ to system memory, every memory access by the processor, whether it’s a memory read, write or execute is tested for its "memory security attributes (i.e. is it a secure or non-secure address).

1.2.1 - SAU relationship with the MPU(s)

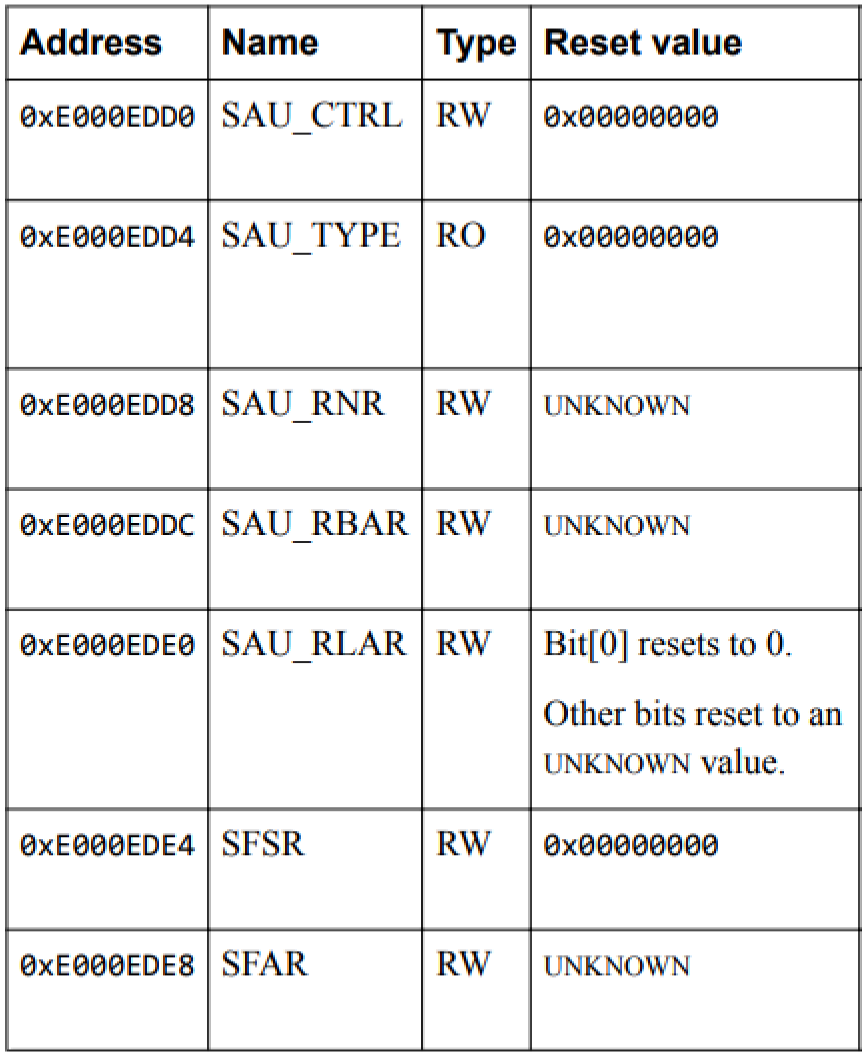
The Memory Protection Unit (MPU) is a programmable unit that allows privileged software to define memory access permission. If the TrustZone is enabled, there can be up to two MPUs, one for Secure and one for Non-secure. So the MPU is banked between Secure and Non-secure states. The number of regions in the Secure and Non-secure MPU can be configured independently and each can be programmed to protect memory for the associated Security state.
While SAU directly enforce secure and non-secure access restrictions, it works with secure and non-secure memory protection units (MPUs) to determine the access rights associated with the target resource.



In order to carry the secure and privilege capabilities over to other memory systems and interfaces, we use logic present in system's bus i.e. the privilege attribute (HPRIV) and secure attribute (HNONSEC) are carried across the internal Advanced High-performance Bus (AHB). This allows extending security to memories and peripherals through bus filters also known as TrustZone-aware peripherals which are directly connected to AHB/APB.

1.2.2 SAU registers

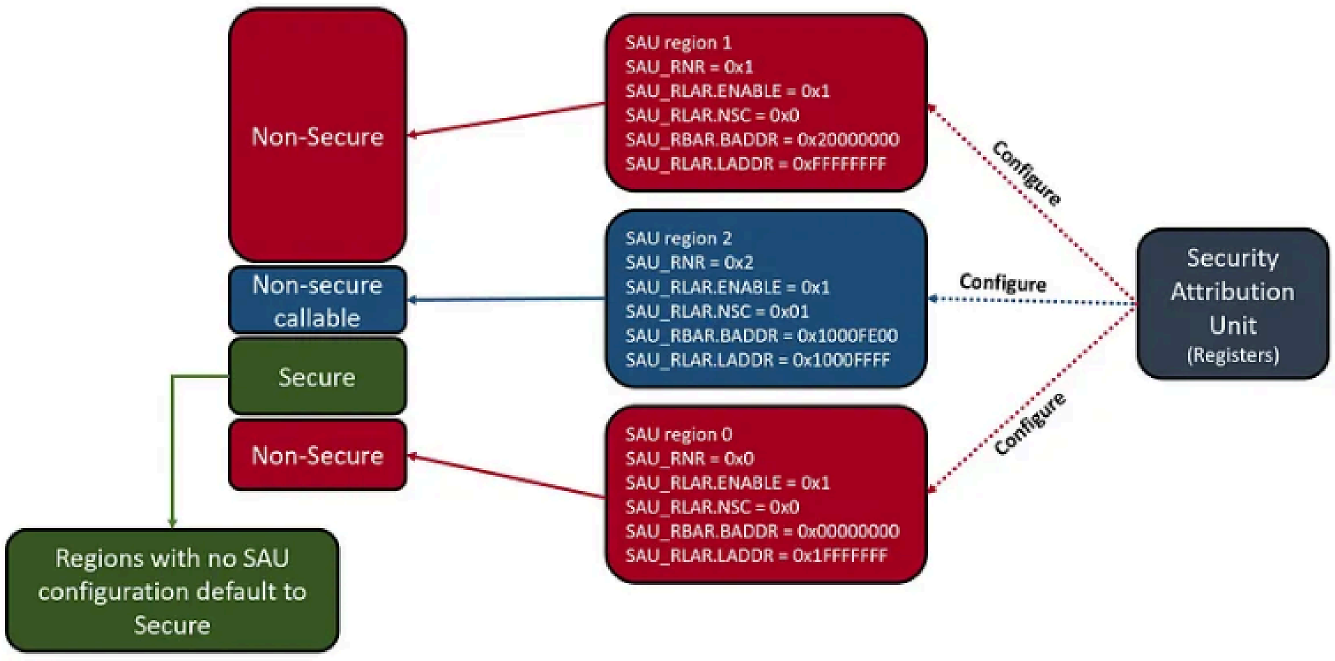
After a power on or reset, an Armv8-M system begins executing code in the secure state. The SAU can be programmed by Secure software through 8 registers.



Address	Name	Type	Reset value
0xE000EDD0	SAU_CTRL	RW	0x00000000
0xE000EDD4	SAU_TYPE	RO	0x00000000
0xE000EDD8	SAU_RNR	RW	UNKNOWN
0xE000EDDC	SAU_RBAR	RW	UNKNOWN
0xE000EDE0	SAU_RLAR	RW	Bit[0] resets to 0. Other bits reset to an UNKNOWN value.
0xE000EDE4	SFSR	RW	0x00000000
0xE000EDE8	SFAR	RW	UNKNOWN

All these registers are in RAZ/WI (Read As Zero, Write Ignore) state when TrustZone is disabled.
These 8 registers enables the attribution of the security regions to memory, determined by a base address and a limit address.
The attribution process of a region is the following:

1. The SAU has to be enabled by a bit in the SAU_CRTL register.
2. The region number of the region that is targeted has to be located in the SAU_TYPE register.
3. In the SAU_RBAR and SAU_RLAR registers the base and limit address of the region have to be placed in order to create the region. The base address must be 32 bit aligned adn the limit address must be a multiple of 32.
4. The security attribute have to be encoded in the SAU_RLAR register.



In the figure an example of regions configuration is shown.

1.2.3 Description of rust SAU driver and Kernel interface. A practical example.

To add the ability to use the SAU within TockOS, it was necessary to create three files: a kernel interface, the implementation of this interface, and a main file to test its functionality.

Kernel Interface:
The file `tock/kernel/src/platform/sau.rs` defines what the Tock kernel expects any SAU implementation to do. The necessary requirements identified are:

- Create a software state that can be assigned to the physical SAU
- Modify this state
- Apply the state to the physical SAU

As a result, it was deemed appropriate to define:

- The attributes that a memory region can have
- The structure of a region
  Furthermore, it was considered suitable to leave the definition of the state to the implementation, as this can vary from system to system.

```
/// tock/kernel/src/platform/sau.rs

/// Possible attribute of a SAU region.
```

```rust
#[derive(Copy, Clone)]
pub enum SauRegionAttribute {
    /// SAU region is Secure
    Secure,
    /// SAU region is Non-Secure Callable
    NonSecureCallable,
    /// SAU region is Non-Secure
    NonSecure,
}

/// Description of a SAU region.
#[derive(Copy, Clone)]
pub struct SauRegion {
    /// First address of the region, its 5 least significant bits must be set
to zero.
    pub base_address: u32,
    /// Last address of the region, its 5 least significant bits must be set
to one.
    pub limit_address: u32,
    /// Attribute of the region.
    pub attribute: SauRegionAttribute,
}

impl SauRegion {
    pub const fn new(base_address: u32, limit_address: u32, attribute:
SauRegionAttribute) -> SauRegion {
        Self {
            base_address: base_address,
            limit_address: limit_address,
            attribute: attribute,
        }
    }
}

pub trait SAU {

    type SauStatus;

    fn enable_sau(&self);
    fn disable_sau(&self);
    fn number_total_regions(&self)-> usize;

    /// None of these five functions write to the hardware
    fn new_status(&self) -> Self::SauStatus;
    fn reset_status(&self, status: &mut Self::SauStatus);
    fn region_is_used(&self, status: &Self::SauStatus, region_number: usize) -
> Option<bool>;
    fn set_region(&self, status: &mut Self::SauStatus, base_address: u32,
limit_address: u32, attribute: SauRegionAttribute, region_number: usize) ->
Option<usize>;
    fn reset_region(&self, status: &mut Self::SauStatus, region_number: usize)
-> Option<usize>;

    /// Commits the status to the hardware
    fn load_status(&mut self, status: &Self::SauStatus) -> Self::SauStatus;

}
```

Implementation of the Interface:
In the file `cortex-v8m/src/sau.rs`, a software copy of the SAU is created, and the kernel interface is implemented to interact with it. The following are realized:

- The software representation of the SAU, in which the registers and the fields of the registers are specified
- Implementation of the software state of the SAU that the kernel will use
- Implementation of the functionalities that the kernel requires for the use of the SAU, as well as its state

```

/// cortex-v8m/src/sau.rs

use kernel::utilities::registers::interfaces::{Readable, Writeable};
use kernel::utilities::registers::{register_bitfields, ReadOnly, ReadWrite};
use kernel::utilities::StaticRef;

use kernel::platform::sau;

pub struct SauRegisters {

    pub sau_ctrl : ReadWrite<u32, SauCtrl::Register>,
    pub sau_type : ReadOnly<u32, SauType::Register>,
    pub sau_rnr  : ReadWrite<u32, SauRnr::Register>,
    pub sau_rbar : ReadWrite<u32, SauRbar::Register>,
    pub sau_rlar : ReadWrite<u32, SauRlar::Register>,
    pub sau_sfsr : ReadOnly<u32, SauSfsr::Register>,
    pub sau_sfar : ReadWrite<u32, SauSfar::Register>,

}

register_bitfields![u32,
    SauCtrl [
        /// When SauCtrl.ENABLE = 0 this bit controls if the memory is marked
as Non-secure
        /// or Secure. If SauCtrl.Enable = 1 this bis has no effect.
        ALLNS     OFFSET(1)   NUMBITS(1) [
            MemIsNonSecure = 1,
            MemIsSecure = 0,
        ],

        /// Enable the SAU.
        ENABLE    OFFSET(0)   NUMBITS(1) [
            Enable = 1,
            Disable = 0
        ]
    ],

    SauType [
        /// The number of implemented SAU regions.
        SREGION  OFFSET(0)   NUMBITS(8) []
    ],

    SauRnr [
        /// Selects the region currently accessed by SAU_RBAR and SAU_RLAR
        REGION OFFSET(0)   NUMBITS(8) []
    ],

    SauRbar [
        /// Upper 27 bits of the lower bound of the selected SAU region.
        /// This value is zero extended to provide the base address.
        BASE    OFFSET(5)   NUMBITS(27) []
    ],

    SauRlar [
        /// Upper 27 bits of the upper bound of the selected SAU region.
        /// This vaule is extended with the value 0x1F (to be sure that the
        /// address is a multiple of 32) to provide the limit address.
        LADDR    OFFSET(5)   NUMBITS(27) [],

        /// Controls whether Non-secure state is permitted to execute an SG
        /// instruction from this region.
        NSC    OFFSET(1)   NUMBITS(1) [
            RegIsNonSecureCallable = 1,
            RegIsNotNonSecureCallable = 0
        ],
        /// SAU region enable.
        ENABLE    OFFSET(0)   NUMBITS(1) [
            Disable = 1,
            Enable = 0
        ]
    ],

    SauSfsr [
        /// Lazy state error flag. Sticky flag indicating that an error
occurred during
        /// lazy state activation or deactivation.

        LSERR    OFFSET(7)    NUMBITS(1) [
            ErrorOccurred = 1,
            ErrorNotOccurred = 0
        ],

        /// This bit is set when the SFAR register contains a valid value.
        SFARVALID    OFFSET(6)    NUMBITS(1) [
            SFARValid = 1,
            SFARNotValid = 0
        ],

        ///
        LSPERR    OFFSET(5)    NUMBITS(1) [
            ErrorOccurred = 1,
            ErrorNotOccurred = 0
        ],

        /// Invalid transition flag. Sticky flag indicating that an exception
was
        /// raised due to a branch that was not flagged as being domain
crossing causing
        /// a transition from Secure to Non-secure memory.
        INVTRAN    OFFSET(4)    NUMBITS(1) [
            ErrorOccurred = 1,
            ErrorNotOccurred = 0
        ],

        /// Attribution unit violation flag. Sticky flag indicating that an
attempt
        /// was made to access parts of the address space that are marked as
Secure
        ///with NS-Req for the transaction set to Non-secure.
        AUVIOL    OFFSET(3)    NUMBITS(1) [
            ErrorOccurred = 1,
            ErrorNotOccurred = 0
        ],

        /// Invalid exception return flag.
        INVER    OFFSET(2)    NUMBITS(1) [
            ErrorOccurred = 1,
            ErrorNotOccurred = 0
        ],

        /// Invalid integrity signature flag. This bit is set if the integrity
signature
        /// in an exception stack frame is found to be invalid during the
unstacking
        /// operation.
        INVIS    OFFSET(1)    NUMBITS(1) [
            ErrorOccurred = 1,
            ErrorNotOccurred = 0
        ],

        /// Invalid entry point. This bit is set if a function call from the
Non-secure
        /// state or exception targets a non-SG instruction in the Secure
state.
        /// This bit is also set if the target address is an SG instruction,
but there
        /// is no matching SAU/IDAU region with the NSC flag set.
        INVEP    OFFSET(0)    NUMBITS(1) [
            ErrorOccurred = 1,
            ErrorNotOccurred = 0
        ],

    ],

    SauSfar [
        /// When the SFARVALID bit of the SFSR is set to 1, this field holds
        /// the address of an access that caused an SAU violation.
        ADDRESS    OFFSET(0)   NUMBITS(32) []
    ],

];

/// Possible error values returned by the SAU methods.
pub enum SauError {
    /// The region number parameter to set or get a region must be between 0
and
    /// region_numbers() - 1.
```

```rust
    RegionNumberTooBig,
    /// Bits 0 to 4 of the base address of a SAU region must be set to zero.
    WrongBaseAddress,
    /// Bits 0 to 4 of the limit address of a SAU region must be set to one.
    WrongLimitAddress,
}


////////////////////////////
//   _____        _    _  //
// /  ____|  /\  | |  | | //
// | (___   /  \ | |  | | //
// \___  \ / /\ \| |  | | //
//  ____) / ____ \ |__| | //
// |_____/_/    \_\____/  //
//                        //
////////////////////////////

/// State related to the real physical SAU.
///
/// There should only be one instantiation of this object as it represents
/// real hardware.

const SAU_BASE_ADDRESS: StaticRef<SauRegisters> =
    unsafe { StaticRef::new(0xE000EDD0 as *const SauRegisters) };

pub struct Sau <const NUM_REGIONS: usize>{

    registers: StaticRef<SauRegisters>,
    }

impl <const NUM_REGIONS: usize> Sau<NUM_REGIONS>{
    pub const unsafe fn new() -> Self {
        Self {
            registers: SAU_BASE_ADDRESS,
        }
    }

    pub fn region_numbers(&self) -> u8 {
       return self.registers.sau_type.read(SauType::SREGION) as u8;
    }

    pub fn enable(&mut self) {
        self.registers.sau_ctrl.write(SauCtrl::ENABLE::Enable)
    }
    pub fn set_region_intern(&mut self, region_number: u8, region:
sau::SauRegion) -> Result<(), SauError> {

        if region_number >= self.region_numbers() {
            Err(SauError::RegionNumberTooBig)
        } else if region.base_address & 0x1F != 0 {
            Err(SauError::WrongBaseAddress)
        } else if region.limit_address & 0x1F != 0x1F {
            Err(SauError::WrongLimitAddress)
        } else {
            self.registers.sau_rnr.set(region_number as u32);

self.registers.sau_rbar.write(SauRbar::BASE.val(region.base_address >> 5));

            match region.attribute {
                sau::SauRegionAttribute::Secure => {

self.registers.sau_rlar.write(SauRlar::LADDR.val(region.limit_address >> 5) +
SauRlar::NSC::RegIsNotNonSecureCallable + SauRlar::ENABLE::Enable);
                }
                sau::SauRegionAttribute::NonSecureCallable => {

self.registers.sau_rlar.write(SauRlar::LADDR.val(region.limit_address >> 5) +
SauRlar::NSC::RegIsNonSecureCallable + SauRlar::ENABLE::Disable);
                }
                sau::SauRegionAttribute::NonSecure => {

self.registers.sau_rlar.write(SauRlar::LADDR.val(region.limit_address >> 5) +
SauRlar::NSC::RegIsNotNonSecureCallable + SauRlar::ENABLE::Disable);
                }
            }
            Ok(())
        }
    }

    pub fn get_region(&mut self, region_number: u8) -> Result<sau::SauRegion,
SauError> {

        if region_number >= self.region_numbers() {
            return Err(SauError::RegionNumberTooBig);
        } else {

self.registers.sau_rnr.write(SauRnr::REGION.val(region_number.into()));
        }

        let base_address = self.registers.sau_rbar.read(SauRbar::BASE);
```

```rust
        let limit_address = self.registers.sau_rlar.read(SauRlar::LADDR);

        let bf = self.registers.sau_rlar.read(SauRlar::NSC) == 1;
        let bf2 = self.registers.sau_rlar.read(SauRlar::ENABLE) == 0;

        let attribute = match (bf, bf2) {
            (_, false) => sau::SauRegionAttribute::Secure,
            (true, true) => sau::SauRegionAttribute::NonSecureCallable,
            (false, true) => sau::SauRegionAttribute::NonSecure,
        };

        Ok(sau::SauRegion {
            base_address : base_address << 5,
            limit_address: (limit_address << 5) | 0x1F,
            attribute,
        })

    }

}


/////////////////////////////////////////////////////
//   _____  _______      _______  _    _  _____    //
// /  ____||__   __| /\ |__   __|| |  | |/  ____|  //
// | (____    | |   /  \   | |   | |  | || (____   //
// \___   \   | |  / /\ \  | |   | |  | |\___   \  //
//  ____) |   | | / ____ \ | |   | |__| | ____) |  //
// |______/   |_|/_/    \_\|_|   \______/|______/  //
//                                                 //
/////////////////////////////////////////////////////

/// SauStatus is the software abstraction of the operational
/// status of the Sau.
/// The cortex-m SAU has eight regions, This struct caches the results
/// of region configuration. Its possible to create a new status, and
/// to modify it. This struct has been created to assign easly a status
/// to the physical SAU. This allows for state manipulation at a different
///time than the physical configuration assignment.

pub struct SauStatus<const NUM_REGIONS: usize> {

    regions: [sau::SauRegion; NUM_REGIONS],
    used: [bool; NUM_REGIONS],
}
```

```
impl<const NUM_REGIONS: usize> SauStatus<NUM_REGIONS> {

    pub fn new() -> Self {
        let regions = [
            sau::SauRegion {
                base_address: 0,
                limit_address: 0,
                attribute: sau::SauRegionAttribute::NonSecure,
            };
            NUM_REGIONS
        ];

        let used = [false; NUM_REGIONS];

        SauStatus { regions, used }
    }

}

//////////////////////////////////////////////////////////
//    _  __                 _   _____         _ _       //
//   | |/ /___ _ _ _ _  ___| | |_   _| _ __ _(_) |_     //
//   | ' </ -_) '_| ' \/ -_) |   | || '_/ _` | |  _|    //
//   |_|\_\___|_| |_||_\___|_|   |_||_| \__,_|_|\__|    //
//                                                      //
//////////////////////////////////////////////////////////

/// SAU module must implement the SAU trait defined inside the Kernel.
/// The Kernel must provide a generic interface that can be specialized for a
specific SAU.
/// The Kernel only implements (in platform/sau.rs) some local structure and
then a large trait.
/// For more details about the trait functions semantics, check in
kernel/src/plaftorm/sau.rs

impl<const NUM_REGIONS: usize> sau::SAU for Sau<NUM_REGIONS> {

    type SauStatus = SauStatus<NUM_REGIONS>;

    fn enable_sau(&self) {
        self.registers.sau_ctrl.write(SauCtrl::ENABLE::Enable)
    }

    fn disable_sau(&self) {
        self.registers.sau_ctrl.write(SauCtrl::ENABLE::Disable)
    }
    fn number_total_regions(&self)-> usize {
        return self.registers.sau_type.read(SauType::SREGION) as usize;
    }


    fn new_status(&self) -> Self::SauStatus {
        let status = Self::SauStatus::new();
        return status;
    }

    fn reset_status(&self, status: &mut Self::SauStatus) {

        for i in 0..NUM_REGIONS {
            status.used[i] = false;
        }
    }


    fn region_is_used(&self, status: &Self::SauStatus, region_number: usize) -
> Option<bool> {
        if region_number < NUM_REGIONS {
            Some(status.used[region_number])
        } else {
            None
        }
    }

    fn set_region(&self, status: &mut Self::SauStatus, base_address: u32,
limit_address: u32, attribute: sau::SauRegionAttribute, region_number: usize)
-> Option<usize> {
        if region_number < NUM_REGIONS {
           status.regions[region_number] = sau::SauRegion {
                base_address: base_address,
                limit_address: limit_address,
                attribute: attribute,
            };
            status.used[region_number] = true;
            Some(region_number)
        } else {
            None
        }
    }

    fn reset_region(&self, status: &mut Self::SauStatus, region_number: usize)
-> Option<usize> {
        if region_number < NUM_REGIONS {
            status.used[region_number] = false;
            status.regions[region_number] = sau::SauRegion {
                base_address: 0,
                limit_address: 0,
                attribute: sau::SauRegionAttribute::NonSecure,
            };
            Some(region_number)
        } else {
            None
        }
    }


    fn load_status(&mut self, status: &Self::SauStatus) -> Self::SauStatus {

        let mut retstatus = Self::SauStatus::new();

        for i in 0..NUM_REGIONS {

           if status.used[i] == true {
               let _ = self.set_region_intern(i as u8, status.regions[i]);
               retstatus.regions[i] = status.regions[i];
           }
        }
        return retstatus;
    }
}
```

Test
The file `nucleol5/src/main.rs` tests the kernel's ability to create a secure/non-secure region. The key steps are:

- Declaration of the region containing the code as secure
- Initialization of a memory location in SRAM with the value `0xFF`
- Declaration of a region containing this SRAM location as secure and reading the value
- Declaration of the same region containing this SRAM location as non-secure and reading the value
- Declaration of the same region containing this SRAM location as secure and reading the value

The goal is to observe how the driver makes the value readable when the SRAM region is declared secure, and how it makes the value non-readable when the SRAM region is declared non-secure. By default, SRAM is declared secure, even when the SAU designates a region as non-secure.

What happens when the region is declared secure:

- The processor generates a secure access, as it executes code in a secure area.
- The SAU does not interfere since the area is declared secure.
- The SRAM does not interfere since it is secure by default.
- The value `0xFF` is read correctly.

What happens when the region is declared non-secure:

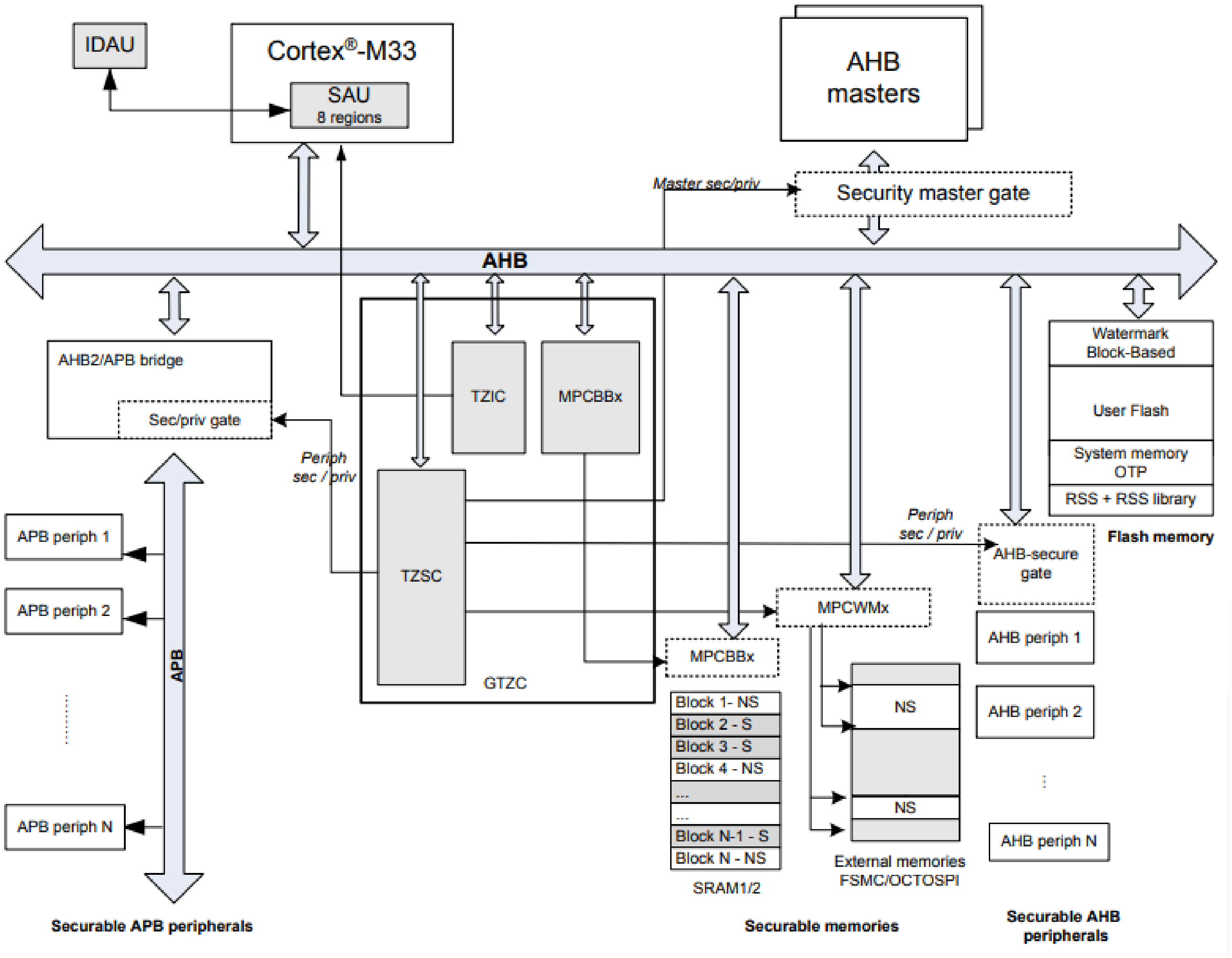
- The processor generates a secure access, as it executes code in a secure area.
- The SAU does not interfere since the area is declared non-secure.
- Since the access passes through the SAU and defines the region as non-secure, the access becomes non-secure.
- The SRAM blocks the read since it is secure by default.
- The value read is `0x00` because a non-secure access to secure SRAM results in a read-at-zero.

```rust
/// nucleol5/src/main.rs

#![no_std]
#![no_main]

use cortex_m_rt::entry;
use panic_halt as _;

use cortex_v8m::sau;
use kernel::platform::sau::SAU;
use kernel::platform::sau::SauRegionAttribute;


#[entry]
unsafe fn main() -> ! {

    // Waiting for *a != 0 from debugger
    let a: *const u32 = 0x2000f000 as *const u32;
    let mut v = unsafe {*a};

    loop {
        v = unsafe{*a};
        if v != 0 {break;}
    }

    // Creating a variable to test the readability of its value if it's in a
secure/non-secure region
    let address: *mut u32 = 0x2000ff00 as *mut u32;
    *address = 0xFF;


    // Sau and status creation
    let mut sau_ref: sau::Sau<8> = sau::Sau::new();
    let mut status: sau::SauStatus<8> = sau_ref.new_status();

    // Setting code region secure and loading the status into the physical sau
    sau_ref.set_region(&mut status, 0x08000000, 0x0805FFFF,
SauRegionAttribute::Secure, 0 as usize);
    sau_ref.load_status(&status);

    // Enabling sau
    sau_ref.enable_sau();

    // -------------------------------------------------------------------- //

    // Setting a SRAM region secure and loading the status into the physical
sau
    sau_ref.set_region(&mut status, 0x2000fa00, 0x2000ffff,
SauRegionAttribute::Secure, 1 as usize);
    sau_ref.load_status(&status);

    // Reading secure SRAM from non-secure code - 0xFF is expected
    let mut value = *address;

    // -------------------------------------------------------------------- //

    // Setting the SRAM region non-secure and loading the status into the
physical sau
    sau_ref.set_region(&mut status, 0x2000fa00, 0x2000ffff,
SauRegionAttribute::NonSecure, 1 as usize);
    sau_ref.load_status(&status);

    // Reading non-secure SRAM from non-secure code - 0x00 is expected
    value = *address;

    // -------------------------------------------------------------------- //

    // Setting the SRAM region secure and loading the status into the physical
sau
```

```
    sau_ref.set_region(&mut status, 0x2000fa00, 0x2000ffff,
SauRegionAttribute::Secure, 1 as usize);
    sau_ref.load_status(&status);

    // Reading non-secure SRAM from non-secure code - 0xFF is expected
    value = *address;

    loop {

    }
}
```

2 - TrustZone on ST32L5 Microcontrollers

The presence of TrustZone is reflected in the systems that surrounds the Cortex-M33, there are also others components that actively contributes to the TrustZone system.



In this image are shown other components then the SAU that contribute to the TrustZone system, as the "implementation defined attribution unit" (IDAU), the Flash memory, and the "global TrustZone security controller" (GTZC). All of these component give their contribuiton to the realization of the TrustZone system outside the mere core.

The default system security state is:

- CPU:
  – Cortex®-M33 is in secure state after reset. The boot address must be in secure address.
- Memory map:
  – SAU: is fully secure after reset. Consequently, all memory map is fully secure. Up to 8 SAU configurable regions are available for security attribution.
- Flash:
  – Flash security area is defined by watermark user options.
  – Flash block based area is non-secure after reset.
- SRAMs:
  – All SRAMs are secure after reset. MPCBB (memory protection block based controller) is secure.
- External memories:
  – FSMC, OCTOSPI banks are secure after reset. MPCWMx (memory protection watermark based controller) are secure.
- Peripherals:
  – Securable peripherals are non-secure after reset.
  – TrustZone-aware peripherals (except the GPIO) are non-secure after reset. Their secure configuration registers are secure. Note: Refer to Table 7 and Table 8 for a list of Securable and TrustZone-aware peripherals.
- All GPIO are secure after reset.
- Interrupts:
  – NVIC: All interrupts are secure after reset. NVIC is banked for secure and non-secure state.
  – TZIC: All illegal access interrupts are disabled after reset.

2.1 - Configuring TrustZone

As said TrustZone is more an ecosystem then a single mechanism. Thus TrustZone is configurable in different ways and this is done by the configuration of the Option-Bytes. TrustZone itself is enabled by one bit of the Option-Bytes.TZEN = 1 enables TrustZone, TZEN = 0 disables it.

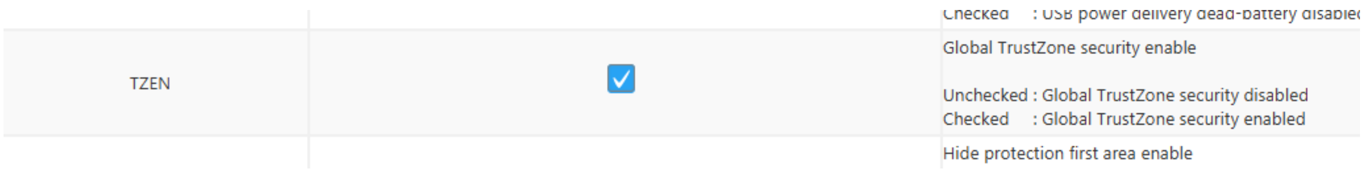
2.1.1 - The Option-Bytes, how it works and what it enables

When an STM32 is powered on, the embedded flash memory automatically loads the Option-Bytes.

- TrustZone Enable (TZEN): enables TrustZone (TZEN = 1)
- Readout protection (RDP) to protect the whole memory. Four levels of protection are available:
  - Level 0: no readout protection.
  - Level 0.5: available only when TrustZone is enabled. All read/write operations (if no write protection is set) from/to the non-secure Flash memory are possible. The Debug access to secure area is prohibited. Debug access to non-secure area remains possible.
  - Level 1: the Flash memory cannot be read from or written to if either the debug features are connected or the boot in RAM or bootloader are selected. If TrustZone is enabled, the non-secure debug is possible and the boot in SRAM is not possible
  - Level 2: the debug features (Cortex®-M33 JTAG and serial wire), the boot in RAM and the bootloader selection are disabled (JTAG fuse). This selection is irreversible.
- At startup, a BOOT0 pin, nBOOT0 and NSBOOTADDx[24:0] / SECBOOTADD0[24:0] option bytes are used to select the boot memory address which includes:
  - Boot from any address in user Flash
  - Boot from system memory bootloader
  - Boot from any address in embedded SRAM
  - Boot from Root Security service (RSS)

2.1.2 - Enable TrustZone, practical execution flow

TrustZone is disabled by default in all STM32L5xx devices. It is activated by setting the TZEN option bit in the FLASH_OPTR register when RDP level is set to Level 0.



		Checked : USB power delivery dead-battery disable
TZEN	✓	Global TrustZone security enable Unchecked : Global TrustZone security disabled Checked : Global TrustZone security enabled
		Hide protection first area enable

This comes from STM32CubeIDE.

2.1.3 - Debug TrustZone execution

Debugging an application in the ARM TrustZone environment is very similar to debugging an application on another ST microcontroller. The only precaution is that debugging can only be performed when the RDP level is set to 0.

2.1.4 - Flash non secure applications and secure applications

A typical project utilizing ARM TrustZone consists of two applications: a secure application and a non-secure application. The process begins with the secure application, which is stored in secure flash and executed at boot. This secure application is responsible for configuring the platform's security settings before calling the non-secure application.
An essential aspect of this setup is that the secure application must implement a non-secure-callable API, allowing the non-secure application to interact with it. The non-secure application, located in non-secure flash, is launched by the secure application and needs to know which functions it can call from the secure side.
Both applications must be compiled into separate binaries, with the secure application being the one that boots first. There are some challenges to address: the non-secure side lacks a traditional main entry point, and when calling functions between the secure and non-secure sides, it's crucial to have an understanding of the available functions, this necessitates the implementation of a vector table to manage these calls effectively.

An alternative way of flashing the applications is linked to the flash programmability:
writing to flash memory by a program is managed by two control registers: one for the secure part and one for the non-secure part, namely FLASH_SECR and FLASH_NSCR. Both registers have a first bit, SECPG and NSPG respectively, that controls the programmability of the flash, which is set to non-programmable by default in both parts. The FLASH_SECR register is programmable only in secure state, while the FLASH_NSCR register is programmable from both secure and non-secure states.
This capability opens the door to an alternative way of creating a secure and a non-secure application within the system: the secure application not only manages the security features of the memory but also takes care of programming the non-secure part in which it can load the non-secure application.

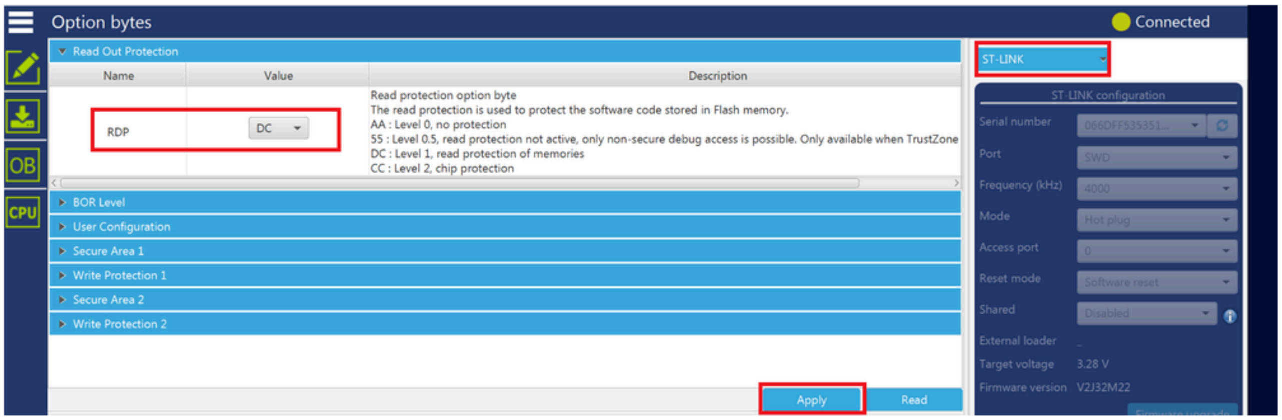
2.1.5 - Disable TrustZone

Once TrustZone is activated on the device, it can only be deactivated during an RDP regression to level 0 (either from RDP level 1 to level 0 or from RDP level 0.5 to level 0). So, disabling TrustZone results in a full chip mass erase: the sample is virgin, corresponding to the production state.
To do TZEN deactivation, the part must already be in RDP level 1 or level 0.5.
when TrustZone® is activated and the RDP level is 0.5 or 1, when the CPU is in secure state, it is not possible to connect to the target through JTAG/SWD, so TZEN/RDP regression is not possible. Only when the CPU is in non-secure state connection to the target through JTAG/SWD and RDP regression are possible.
So if TrustZone is enabled and if non-secure code cannot be executed, maybe because it does

not exist, the CPU always remains in the secure state and the RDP regression cannot be done through JTAG / SWD in such cases. So two different procedures can take place, depending on the possibility to call or not the non-secure code.

Now the procedure with non-secure code presence is explained.

1. Make sure that the loaded code has both secure and non-secure code and that non-secure code can be called from secure code.
2. Set RDP level to level 1.



Option bytes — Connected

Read Out Protection

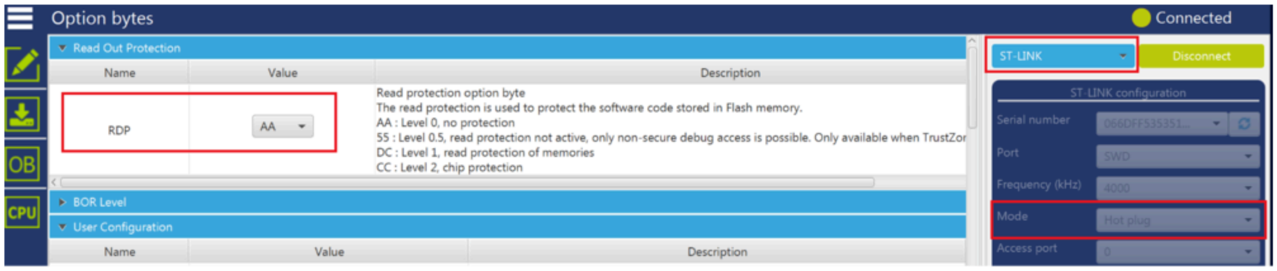
Name	Value	Description
RDP	DC	Read protection option byte The read protection is used to protect the software code stored in Flash memory. AA : Level 0, no protection 55 : Level 0.5, read protection not active, only non-secure debug access is possible. Only available when TrustZone DC : Level 1, read protection of memories CC : Level 2, chip protection

BOR Level
User Configuration
Secure Area 1
Write Protection 1
Secure Area 2
Write Protection 2

Apply | Read

ST-LINK configuration
Serial number: 066DFF535351...
Port: SWD
Frequency (kHz): 4000
Mode: Hot plug
Access port: 0
Reset mode: Software reset
Shared: Disabled
External loader: _
Target voltage: 3.28 V
Firmware version: V2J32M22

3. Do TZEN and RDP regression.



Option bytes — Connected

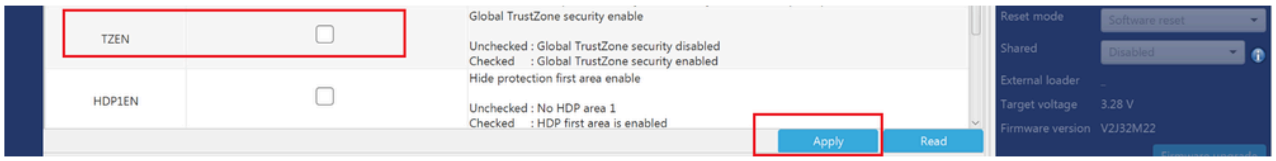
ST-LINK | Disconnect

Read Out Protection

Name	Value	Description
RDP	AA	Read protection option byte The read protection is used to protect the software code stored in Flash memory. AA : Level 0, no protection 55 : Level 0.5, read protection not active, only non-secure debug access is possible. Only available when TrustZor DC : Level 1, read protection of memories CC : Level 2, chip protection

BOR Level
User Configuration

ST-LINK configuration
Serial number: 066DFF535351...
Port: SWD
Frequency (kHz): 4000
Mode: Hot plug
Access port: 0

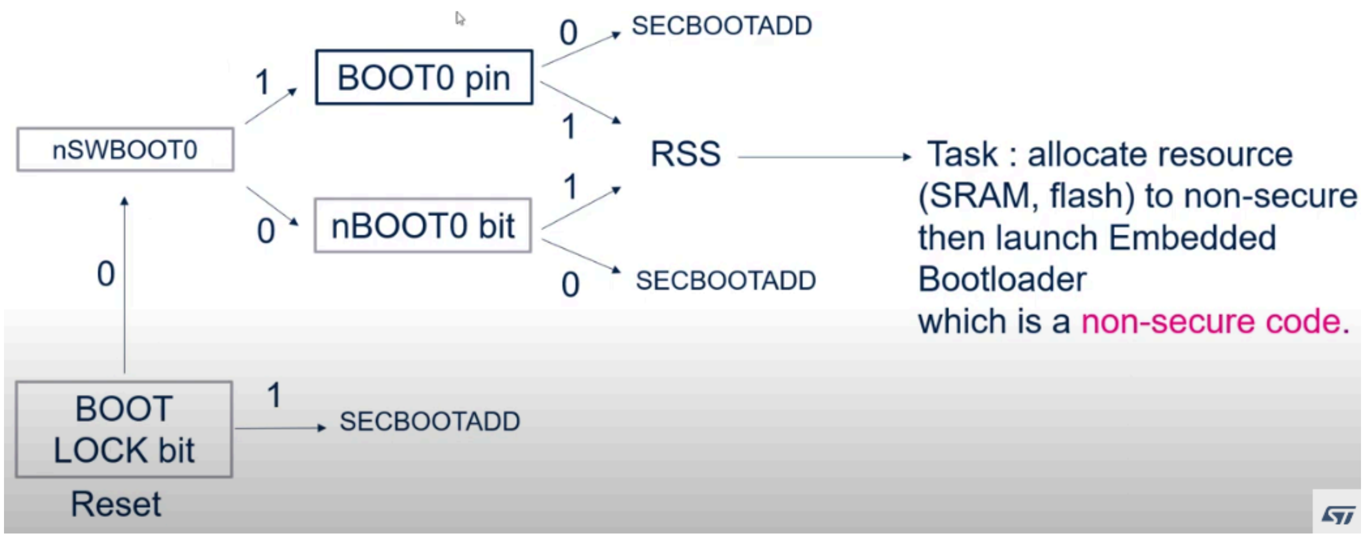


Name	Value	Description
TZEN	☐	Global TrustZone security enable Unchecked : Global TrustZone security disabled Checked : Global TrustZone security enabled
HDP1EN	☐	Hide protection first area enable Unchecked : No HDP area 1 Checked : HDP first area is enabled

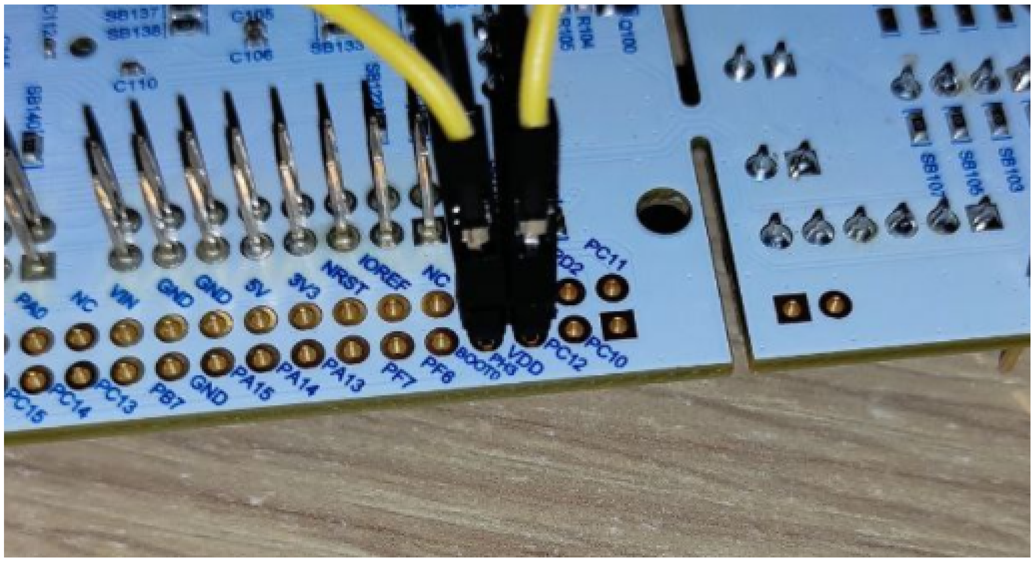
Apply | Read

Reset mode: Software reset
Shared: Disabled
External loader: _
Target voltage: 3.28 V
Firmware version: V2J32M22

Now the procedure without a functioning secure/non-secure code.
As said the core must be in non-secure state to be able to connect, but without the possibility to call the non-secure code this cannot be done. The only way is to force the core to execute non-secure code. This can be done by executing the Root Security System (RSS) that is a piece of code in the FLASH that allocate resources (SRAM, FLASH) to non-secure and launche the Embedded Bootloader which is a non-secure code. In order to do so different option bytes have to be correctly configured, as shown in the figure:


nSWBOOT0
1
BOOT0 pin
0
SECBOOTADD
1
RSS
Task : allocate resource (SRAM, flash) to non-secure then launch Embedded Bootloader which is a non-secure code.
0
nBOOT0 bit
1
0
SECBOOTADD
0
BOOT LOCK bit
1
SECBOOTADD
Reset

When the Reset is pressed option bytes are loaded from the FLASH, so if BOOTLOCK = 0 and nSWBOOT0 = 1 and BOOT0 pin is high the core will execute in non-secure state.
Its important to note that BOOT0 pin has to be driven to Vdd as shown:



So the seps are:

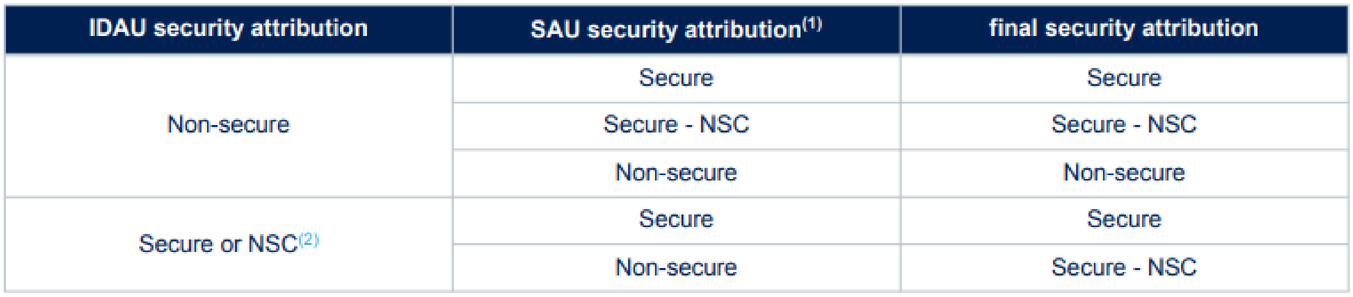
1. make sure that the option bytes are properly configured as in the figure.
2. connect BOOT0 pin to Vdd.
3. Set RDP level to level 0.5.
4. Do TZEN and RDP regression.

2.2 - The IDAU

As previously mentioned the "implementation defined attritution unit" (IDAU), in combination with the SAU, controls the security state of a memory address.
The IDAU defines a memory-map partition that is not configurable, indeed, as the name "implentation defined attritution unit" suggests, the memory-map partition is defined by default by the producer of the microcontroller, in this case STM.

2.2.1 - IDAU relationship with the SAU

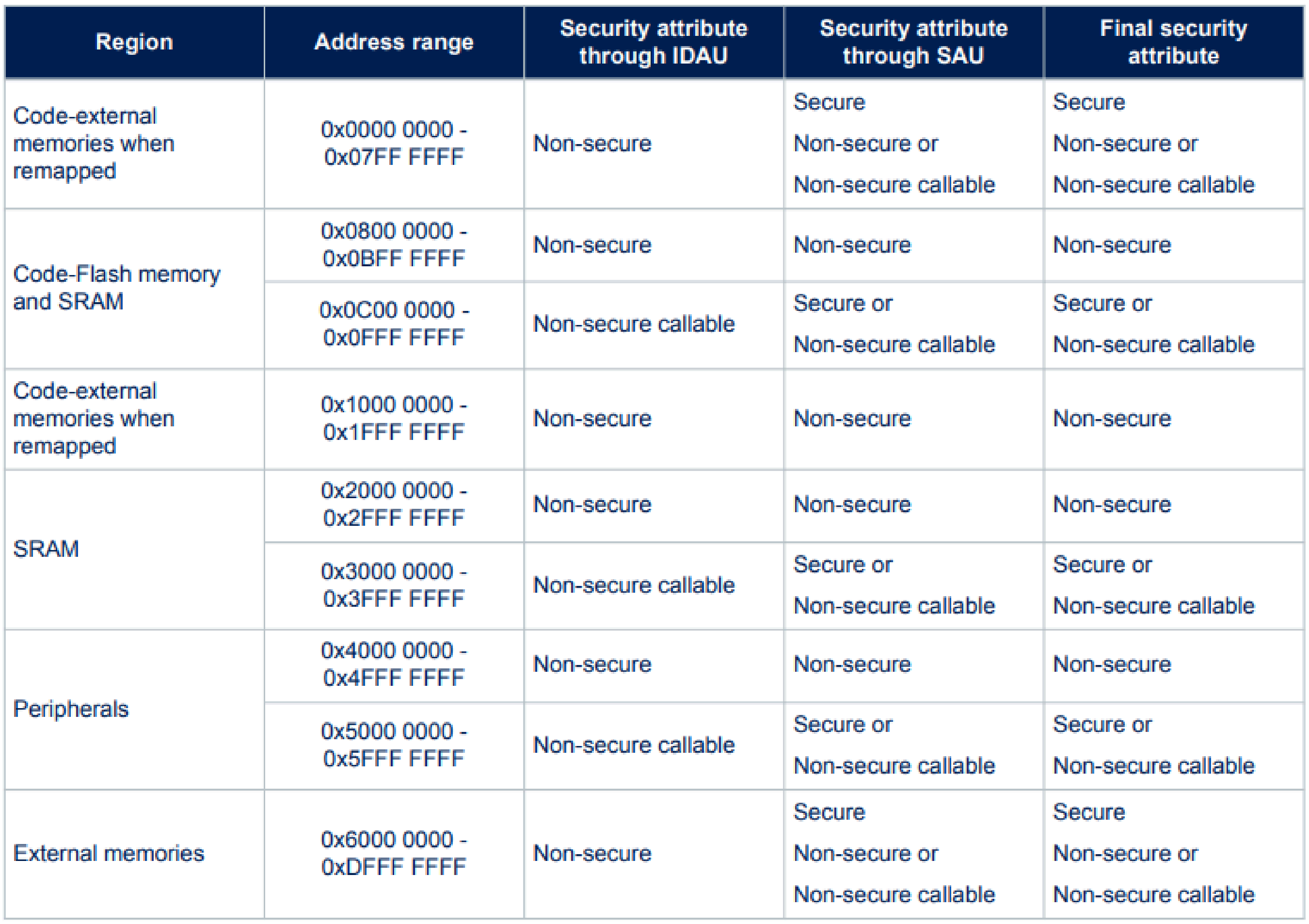
The final security attribute of an address is defined by the most restrionction between the SAU and the IDAU, as shown in figure:



IDAU security attribution	SAU security attribution[1]	final security attribution
Non-secure	Secure	Secure
	Secure - NSC	Secure - NSC
	Non-secure	Non-secure
Secure or NSC[2]	Secure	Secure
	Non-secure	Secure - NSC

2.2.2 - Microcontroller Memory Map

As said the IDAU defines a memory-map division in different regions with different secure attributes. This map has to be conbined with the map defined by the SAU. The final result is:



Region	Address range	Security attribute through IDAU	Security attribute through SAU	Final security attribute
Code-external memories when remapped	0x0000 0000 - 0x07FF FFFF	Non-secure	Secure Non-secure or Non-secure callable	Secure Non-secure or Non-secure callable
Code-Flash memory and SRAM	0x0800 0000 - 0x0BFF FFFF	Non-secure	Non-secure	Non-secure
	0x0C00 0000 - 0x0FFF FFFF	Non-secure callable	Secure or Non-secure callable	Secure or Non-secure callable
Code-external memories when remapped	0x1000 0000 - 0x1FFF FFFF	Non-secure	Non-secure	Non-secure
SRAM	0x2000 0000 - 0x2FFF FFFF	Non-secure	Non-secure	Non-secure
	0x3000 0000 - 0x3FFF FFFF	Non-secure callable	Secure or Non-secure callable	Secure or Non-secure callable
Peripherals	0x4000 0000 - 0x4FFF FFFF	Non-secure	Non-secure	Non-secure
	0x5000 0000 - 0x5FFF FFFF	Non-secure callable	Secure or Non-secure callable	Secure or Non-secure callable
External memories	0x6000 0000 - 0xDFFF FFFF	Non-secure	Secure Non-secure or Non-secure callable	Secure Non-secure or Non-secure callable

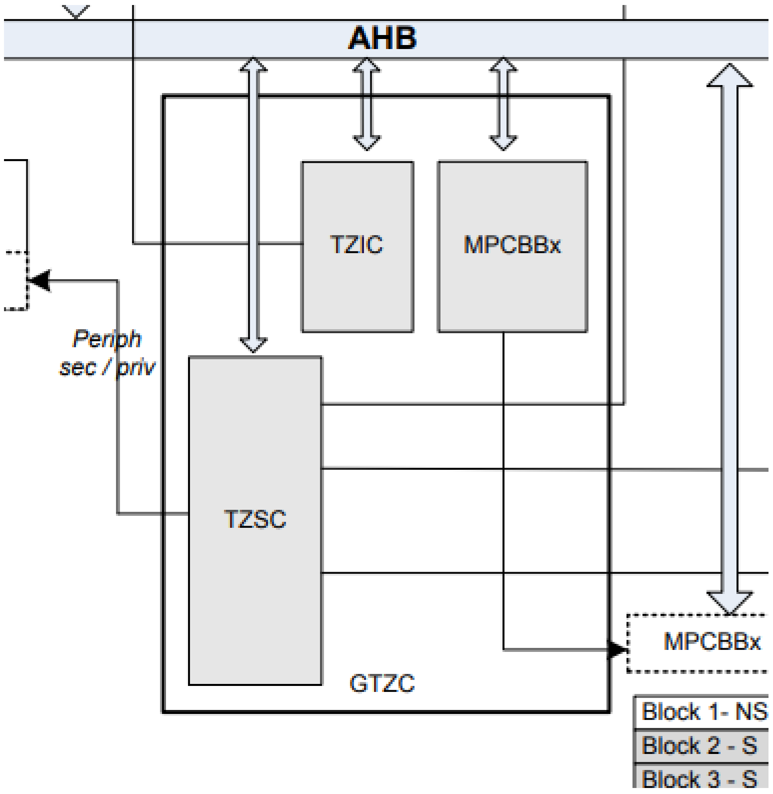
2.3 - TrustZone Soc peripherals

In addition to the Cortex-M33 TrustZone feature, the STM32L5 devices come with complementary security features that reinforce and allow a more flexible partition between the secure and the non-secure worlds, by providing a second level of security on top of the SAU/IDAU.

2.3.1 - GTZC behaviour, role and purpose

The "Global TrsutZone controller" (GTZC) has the following subblocks:

- TZSC (TrustZone security controller) allows the security attribute configuration of:
  – peripherals (see the note below) as either secure or non-secure
  – external memories: through watermark-memory-protection controller (MPCWMx, x = 1, 2, 3)
- MPCBBx (block-based memory protection controller) allows the security attribute configuration of SRAM blocks as follows. The SRAM1 and SRAM2 can be programmed as secure or non-secure by block-based using the MPCBB. The granularity of SRAM secure block-based is a page of 256 bytes.
- TZIC (TrustZone illegal access controller) gathers all illegal access events in the system, and generates a secure interrupt towards the NVIC (GTZC_IRQn).



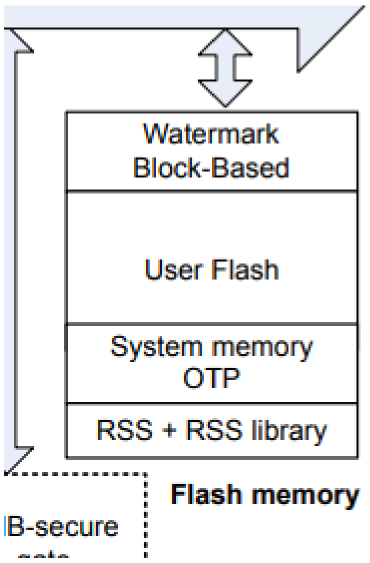
When the TrustZone security is active, a peripheral is either securable or TrustZone-aware:

- Securable: the security attribute is configured by GTZC/TZSC controller.

- TrustZone-aware: the security attribute is configured using some peripheral secure registers. For example, the GPIO is TrustZone-aware with a security attribute configured through the GPIOx_SECCFGR secure register.

2.3.2 - Flash TrustZone extension behaviour, role and purpose

In the STM system is possible to extend the secuiry mechanism control with another indipendent way. To secure the Flash memory STM gives the possibility to use into-Flash controllers that filters the memory requests besed on security attributes. This is realized by two different types of mechanisms: Watermark-based and Block-Based.



- Block-Based: Using a block-based gate flash can be divided into multiple, alternating blocks of secure and non-secure regions. A block-based gate filters transactions based on these regions.
- A Watermark-Based gate in contrast to the block-based gate splits flash or into two regions divided by a watermark address, one region being secure and the other non-secure.

The Watermark-Based is programmable through two registers in the Option-Bytes of the Flash, SECWMx_PSTRT and SECWMx_PEND (x = 1, 2). Where the former is the start address of the region and the latter is the end address of the region.
Note that Block-based registers can only set a page as secure whereas it is set as non-secure through Flash secure watermark option bytes. The opposite is not possible: a page cannot be configured as non-secure using block-based registers, when it is configured as secure through Flash secure watermark option bytes.