life.augmented

MPU usage in STM32 with ARM Cortex®-M7

T.O.M.A.S. team

Hello, and welcome to this presentation about MPU usage in STM32. With ARM Cortex-M7.

Agenda

The purpose of the presentation is to show usage and setting of MPU on STM32 with Cortex-M7 and mainly **raise awareness of the issue with speculative access on Cortex-M7, which may cause speculative read lock, and the issue may be prevented by MPU.**

Presentation covers also basic parameters of MPU and options we can set.

At the end you can find also few typical examples of MPU setting.

Presentation is not covering security aspect of MPU usage like setting application permissions to access only some part of memory.

Only purpose of MPU usage in this presentation is to make project run reliably and with best possible performance.

Cortex®-M7 speculative read

- Description
  - The Cortex®-M7 does some speculative read accesses to normal memory regions. These speculative read accesses could cause high latency or system errors when performed on external memories like SDRAM or Quad-SPI, which don’t need to be connected or have such size.

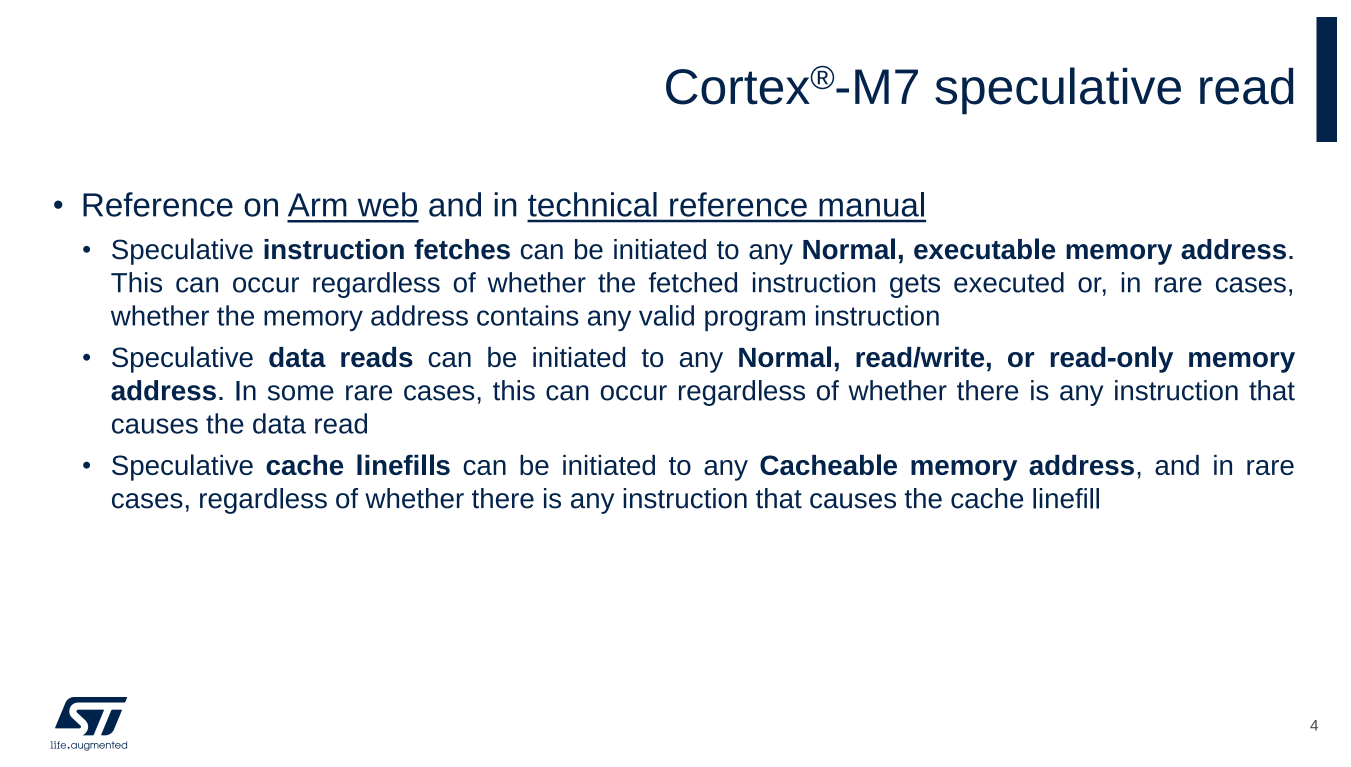
We start with the description of Cortex-M7 speculative read feature.

Speculative memory read may be performed by Cortex-M7 core on Normal memory regions.

Purpose of speculative read is to increase performance of the microcontroller.

Speculative memory read may cause high latency or even system error when performed on external memories like SDRAM, or Quad-SPI.

External memories even don't need to be connected to microcontroller, but its memory range is accessible by speculative read because by default, its memory region is set as Normal.



Cortex®-M7 speculative read

- Reference on Arm web and in technical reference manual
  - Speculative **instruction fetches** can be initiated to any **Normal, executable memory address**. This can occur regardless of whether the fetched instruction gets executed or, in rare cases, whether the memory address contains any valid program instruction
  - Speculative **data reads** can be initiated to any **Normal, read/write, or read-only memory address**. In some rare cases, this can occur regardless of whether there is any instruction that causes the data read
  - Speculative **cache linefills** can be initiated to any **Cacheable memory address**, and in rare cases, regardless of whether there is any instruction that causes the cache linefill

ARM technical reference manual exactly lists situations when speculative access may be done by core.

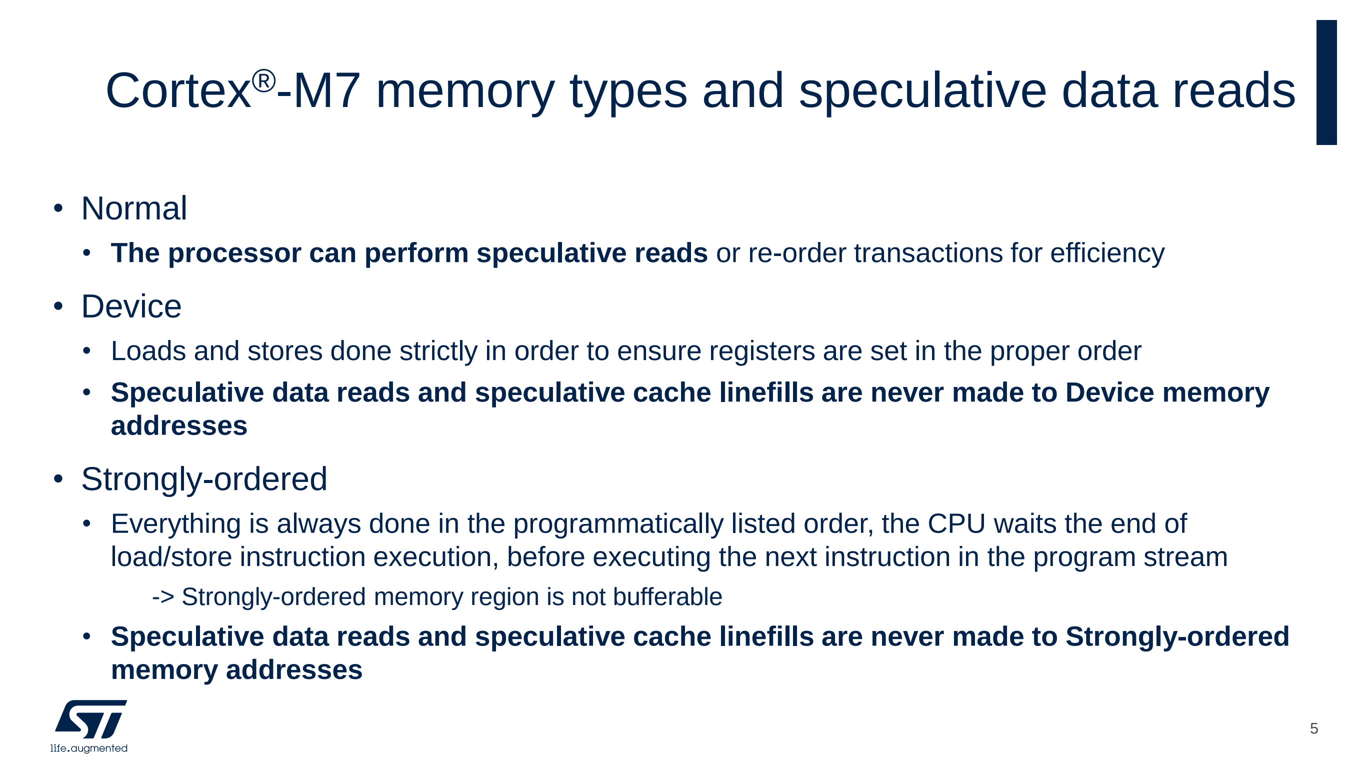
Speculative access cannot be predicted.

It's possible to disable speculative access in core registers, but due to performance drop, this option is not recommended.

Speculative instruction fetches can be initiated to any normal executable memory address. This can occur regardless of whether the fetched instruction gets executed or in rare cases, whether the memory address contains any valid program instruction.

Speculative data reads can be initiated to any normal read, write or read only memory address. In some rare cases, this can occur regardless of whether there is any instruction that causes the data read.

Speculative cache linefill can be initiated to any cacheable memory address and in rare cases, regardless of whether there is any instruction that causes the cache linefill.



Cortex®-M7 memory types and speculative data reads

- Normal
  - **The processor can perform speculative reads** or re-order transactions for efficiency
- Device
  - Loads and stores done strictly in order to ensure registers are set in the proper order
  - **Speculative data reads and speculative cache linefills are never made to Device memory addresses**
- Strongly-ordered
  - Everything is always done in the programmatically listed order, the CPU waits the end of load/store instruction execution, before executing the next instruction in the program stream

    -> Strongly-ordered memory region is not bufferable
  - **Speculative data reads and speculative cache linefills are never made to Strongly-ordered memory addresses**

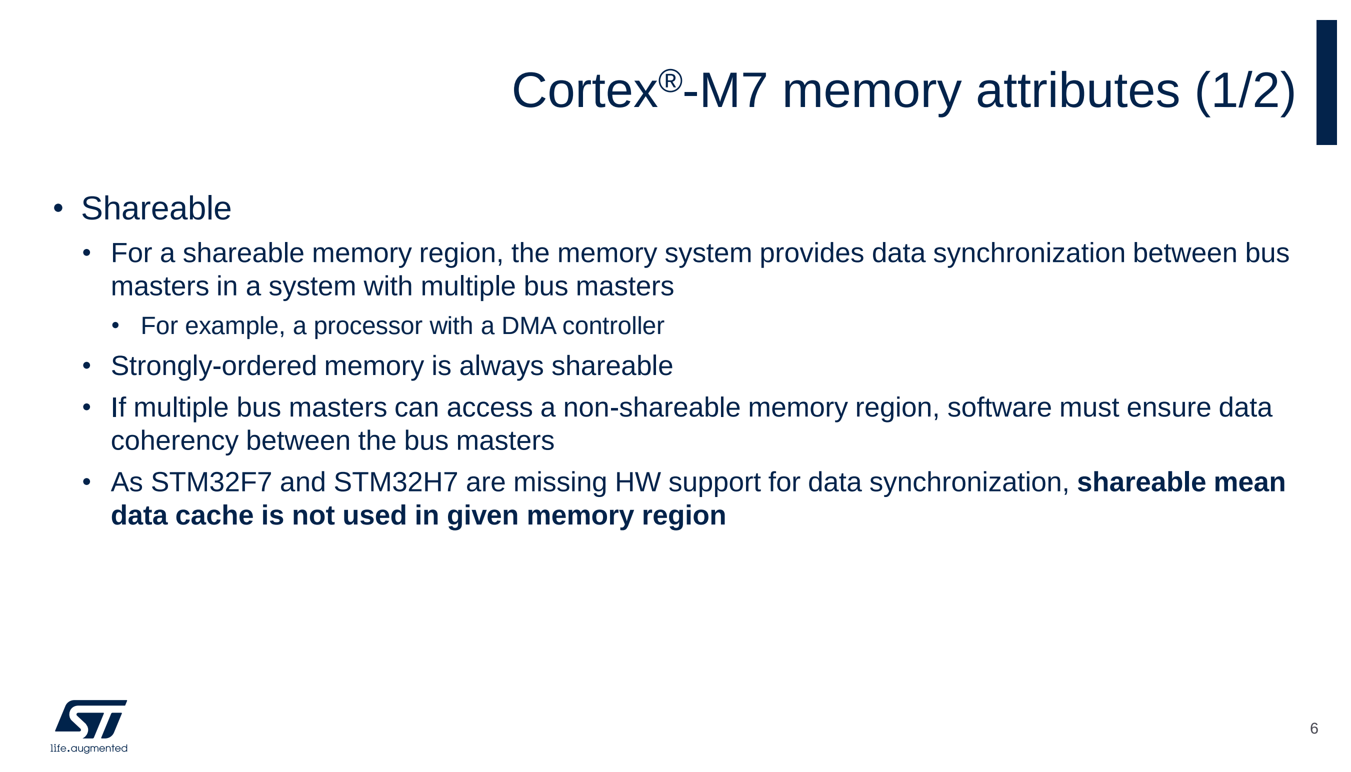
There are three types of memory regions for Cortex-M7 devices.
Memory type determines which operations are allowed on given memory region.

In **Normal** memory regions processor can perform speculative reads or re-order transactions for efficiency. Processor can also perform unaligned memory access, so **[Normal] memory type is convenient for code execution**.

Both **Device** memory type and **Strongly Ordered** memory type do load and store operations **strictly in program order**.
Difference is, that **Device memory type is bufferable** - meaning that instruction execution may continue before memory right is done.
Memory write is then finished from a buffer.

For **Strongly Ordered** memory region **CPU waits for the end of memory access instruction.**

Speculative access is never made to Strongly Ordered and **Device** memory areas.
Device memory type is used for microcontroller registers.
NOT_BUFFERED, Strongly Ordered type is used for memories where each writes need to be visible for devices - for example, for external NAND memories.



Cortex®-M7 memory attributes (1/2)

- Shareable
  - For a shareable memory region, the memory system provides data synchronization between bus masters in a system with multiple bus masters
    - For example, a processor with a DMA controller
  - Strongly-ordered memory is always shareable
  - If multiple bus masters can access a non-shareable memory region, software must ensure data coherency between the bus masters
  - As STM32F7 and STM32H7 are missing HW support for data synchronization, **shareable mean data cache is not used in given memory region**

life.augmented

In ARM Cortex-M devices there are additionally two attributes to be set for each memory region:

Shareable shall be set for a region if multiple masters can access the region, and it is up to the memory system to provide data synchronization between multiple masters. Typical example of multiple masters accessing same memory in STM32 is processor core and DMA. In this case, data cache may cause different data to be visible for core and for DMA.

As STM 32 [F7 and H7] microcontrollers don't contain any hardware feature for keeping data coherent, setting a region as Shareable means that data cache is not used in the region.
If region is not shareable, data cache can be used, but data coherency between bus masters need to be ensured by software.

Cortex®-M7 memory attributes (2/2)

- Execute Never (XN)
  - Means the processor prevents instruction accesses. A HardFault exception is generated on executing an instruction fetched from an XN region of memory
  - **Speculative instruction fetches are never made to memory addresses in an Execute Never region**

Second attribute is Execute Never:

When Execute Never is set for a region, instructions cannot be executed from that region, and any attempt for that causes hard fault.

This attribute has more usage in security usage of MPU.

For this presentation it is more important that *speculative instruction fetch* cannot be done in Execute Never region.

Cortex®-M7 memory region

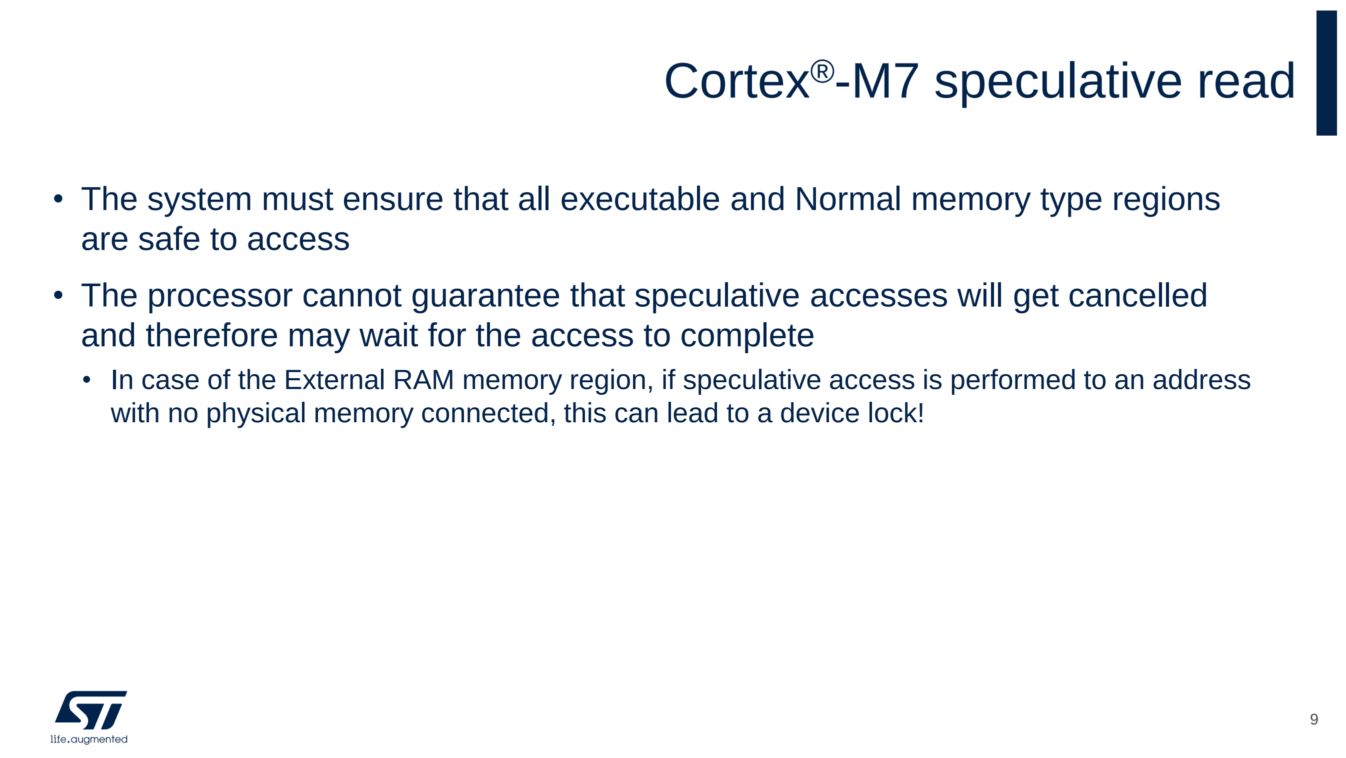
Address range	Memory region	Memory type	XN
0x00000000-0x1FFFFFFF	Code	Normal	-
0x20000000-0x3FFFFFFF	SRAM	Normal	-
0x40000000-0x5FFFFFFF	Peripheral	Device	XN
0x60000000-0x9FFFFFFF	**External RAM**	**Normal**	**-**
0xA0000000-0xDFFFFFFF	External device	Device	XN
0xE0000000-0xE00FFFFF	Private Peripheral Bus	Strongly- ordered	XN
0xE0100000-0xFFFFFFFF	Vendor-specific device	Device	XN

life.augmented

Table on this slide shows complete address range of Cortex-M7 devices and default memory types in given region.

Memory access in the regions follows rules of its memory type default memory type setting for each region or its part can be changed using MPU.

For speculative access issue, it is important to note that external RAM region is by default Normal memory type with enabled code execution, but



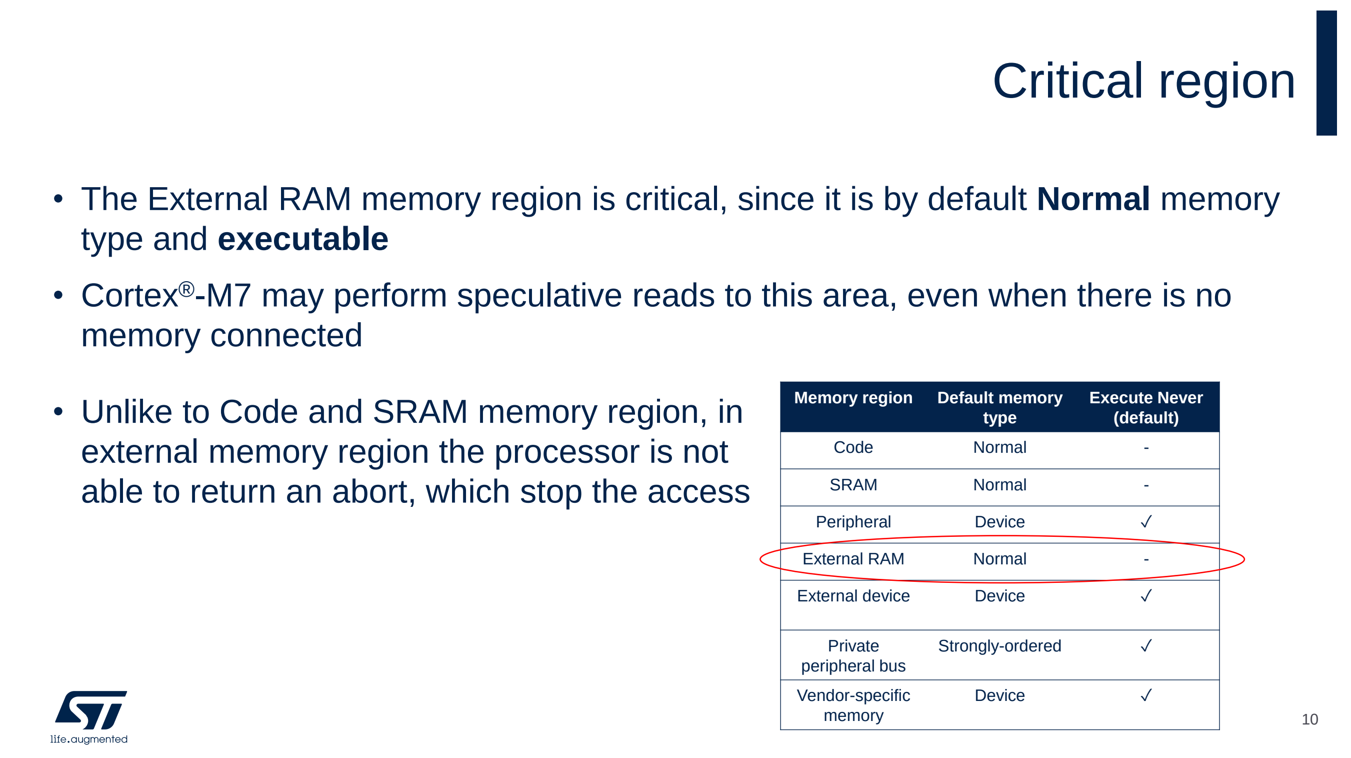
Cortex®-M7 speculative read

- The system must ensure that all executable and Normal memory type regions are safe to access
- The processor cannot guarantee that speculative accesses will get cancelled and therefore may wait for the access to complete
  - In case of the External RAM memory region, if speculative access is performed to an address with no physical memory connected, this can lead to a device lock!

life.augmented

system must ensure that all Executable and Normal memory type regions are safe to access.

If any inaccessible memory location is addressed by speculative access, processor cannot guarantee cancellation of such speculative read, which may lead to extensive delay or even to device lock.



Critical region

- The External RAM memory region is critical, since it is by default **Normal** memory type and **executable**
- Cortex®-M7 may perform speculative reads to this area, even when there is no memory connected
- Unlike to Code and SRAM memory region, in external memory region the processor is not able to return an abort, which stop the access

Memory region	Default memory type	Execute Never (default)
Code	Normal	-
SRAM	Normal	-
Peripheral	Device	✓
External RAM	Normal	-
External device	Device	✓
Private peripheral bus	Strongly-ordered	✓
Vendor-specific memory	Device	✓

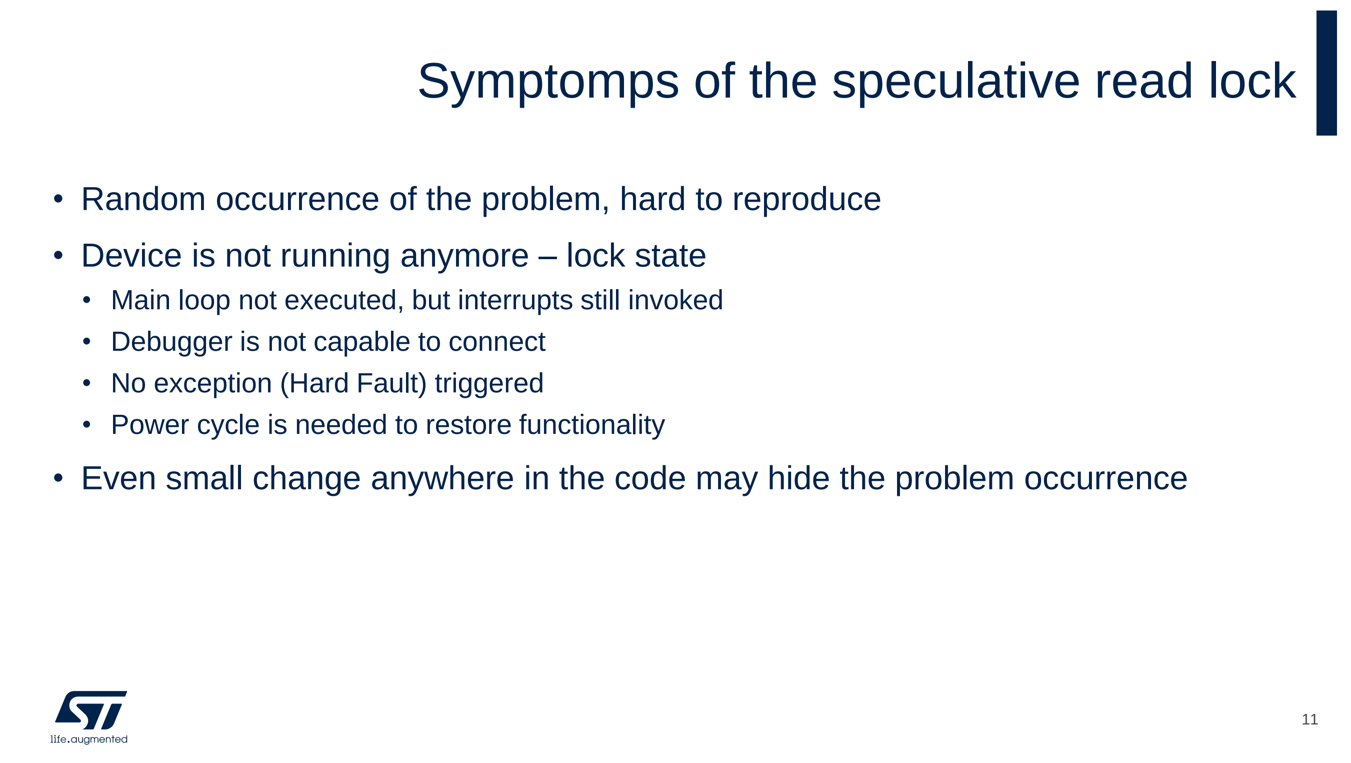
In default mapping of Cortex-M7 devices, external RAM memory region is critical: it's type is Normal, without Executed Never attribute set.

So speculative access can be performed to this region, but external memories
- don’t have to be connected, or
- don't have size to cover complete region size, or
- range not covered by external memory,

shall get their memory type changed by MPU setting to prevent speculative read issue.

External memory may have Normal memory type or any other type convenient for such memory type.

Unlike the external RAM region, both Code and SRAM region are safe to access. Microcontroller memory driver handles this memory range.



Symptomps of the speculative read lock

- Random occurrence of the problem, hard to reproduce
- Device is not running anymore – lock state
  - Main loop not executed, but interrupts still invoked
  - Debugger is not capable to connect
  - No exception (Hard Fault) triggered
  - Power cycle is needed to restore functionality
- Even small change anywhere in the code may hide the problem occurrence

life.augmented

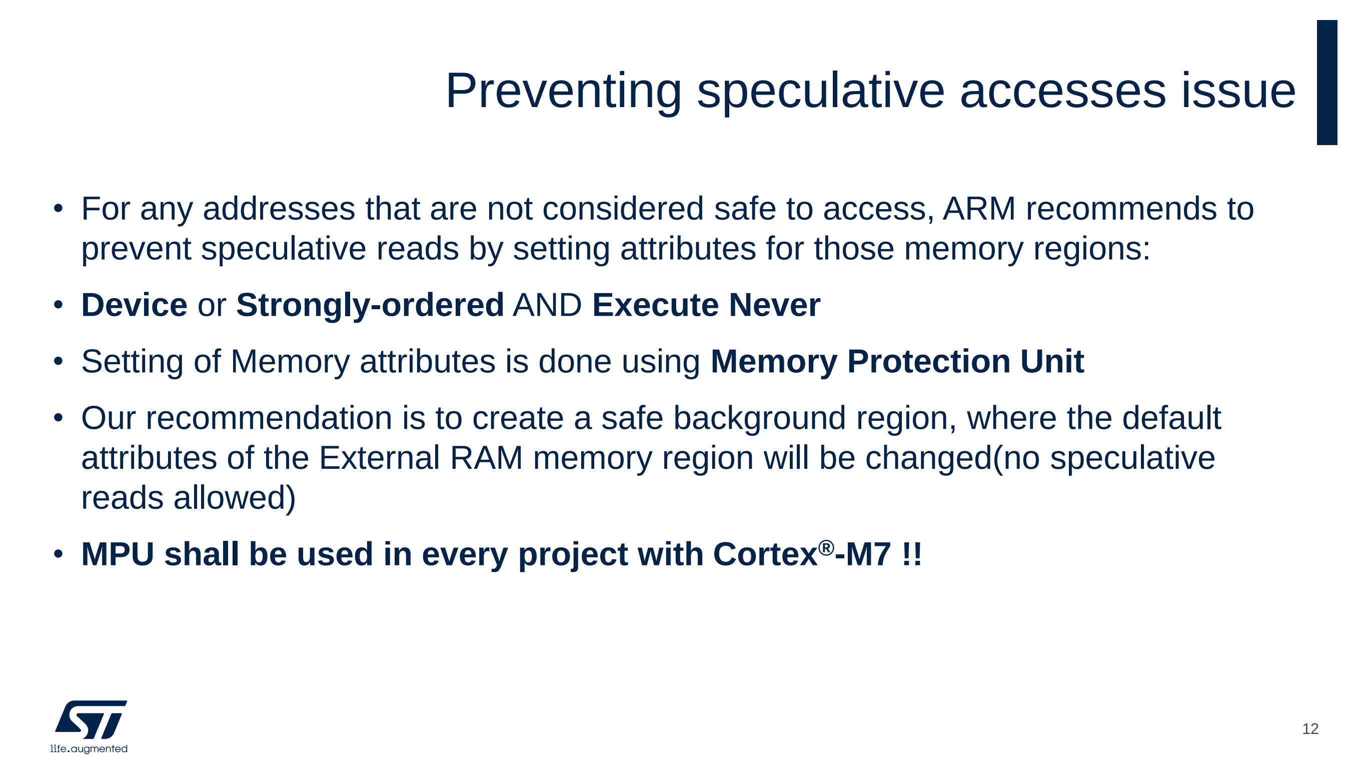
How to recognize the issue:

Typically, if microcontroller got locked due to speculative read, program main loop is not executed anymore, but interrupts are still invoked, no hard fault is triggered.

If microcontroller was in Debug Mode, debug session fails and it's not possible to connect again not even using connect to running target devices not responding on reset. Power cycle is needed.

Occurrence of the issue is random. Even very small change in the code may hide or release the problem.

For example, one luck instruction can determine if the lock will occur, or not.



Preventing speculative accesses issue

- For any addresses that are not considered safe to access, ARM recommends to prevent speculative reads by setting attributes for those memory regions:
- **Device** or **Strongly-ordered** AND **Execute Never**
- Setting of Memory attributes is done using **Memory Protection Unit**
- Our recommendation is to create a safe background region, where the default attributes of the External RAM memory region will be changed(no speculative reads allowed)
- **MPU shall be used in every project with Cortex®-M7 !!**

To prevent speculative access issue, all addresses not safe to access shall change memory type to

- Device or
- Strongly Ordered memory type AND set Execute Never attribute.

After that device cannot be locked by speculative memory access.

To make such setting of memory regions Cortex-M7 contains memory protection unit alias MPU.

On following slides we will introduce code for setting safe background region to set attributes for critical region.

We recommend to use MPU for handling critical external RAM region in each project which is using STM32 based on Cortex-M7.

Memory protection unit

- Modify default memory attributes and access permissions for involved memory areas
- MPU monitors bus transactions
- Number of possible MPU regions depends on a particular device
- When an access rule is violated, a fault exception is triggered
- Overlapping regions – if two regions are setting same memory area, setting from region with higher number is used

life.augmented

Memory protection unit is part of microcontroller core.

Its task is to define memory attributes and access permissions.
MPU is then monitoring bus transaction and if any rule violation is detected, fault exception is triggered.

In Cortex-M7, users see memory management handle the trigger when MPU rules are violated. But for example in Cortex-M0+, memory management handler is not available and MPU fault triggers hard fault.

Depending on used device, also number of region, which can be defined in MPU, is changing.

We need to highlight also MPU behavior with overlapping regions which we will use further in this presentation.
If some memory area is covered by more MPU regions, region with the highest number is used for setting attributes on the memory address.
We set region 0 to prevent speculative read issue, but any other region will get priority over region 0 to propagate preferred rules on used memory.

MPU parameters

- Function HAL_MPU_Enable has 1 argument, which has 4 setting options
  - Options are combining all setting of two bits in MPU_CTRL register
  - HFNMIENA – enables MPU operation in fault handlers
  - PRIVDEFENA – Enable or disable use of default memory map for privileged access

`MPU_HFNMI_PRIVDEF_NONE` –	disabled MPU in fault, disabled default memory map
`MPU_HARDFAULT_NMI` –	enabled MPU in fault, disabled default memory map
`MPU_PRIVILEGED_DEFAULT` –	disabled MPU in fault, enabled default memory map
`MPU_HFNMI_PRIVDEF` –	enabled MPU in fault, enabled default memory map

- Examples in this presentation are done with `MPU_PRIVILEGED_DEFAULT` setting

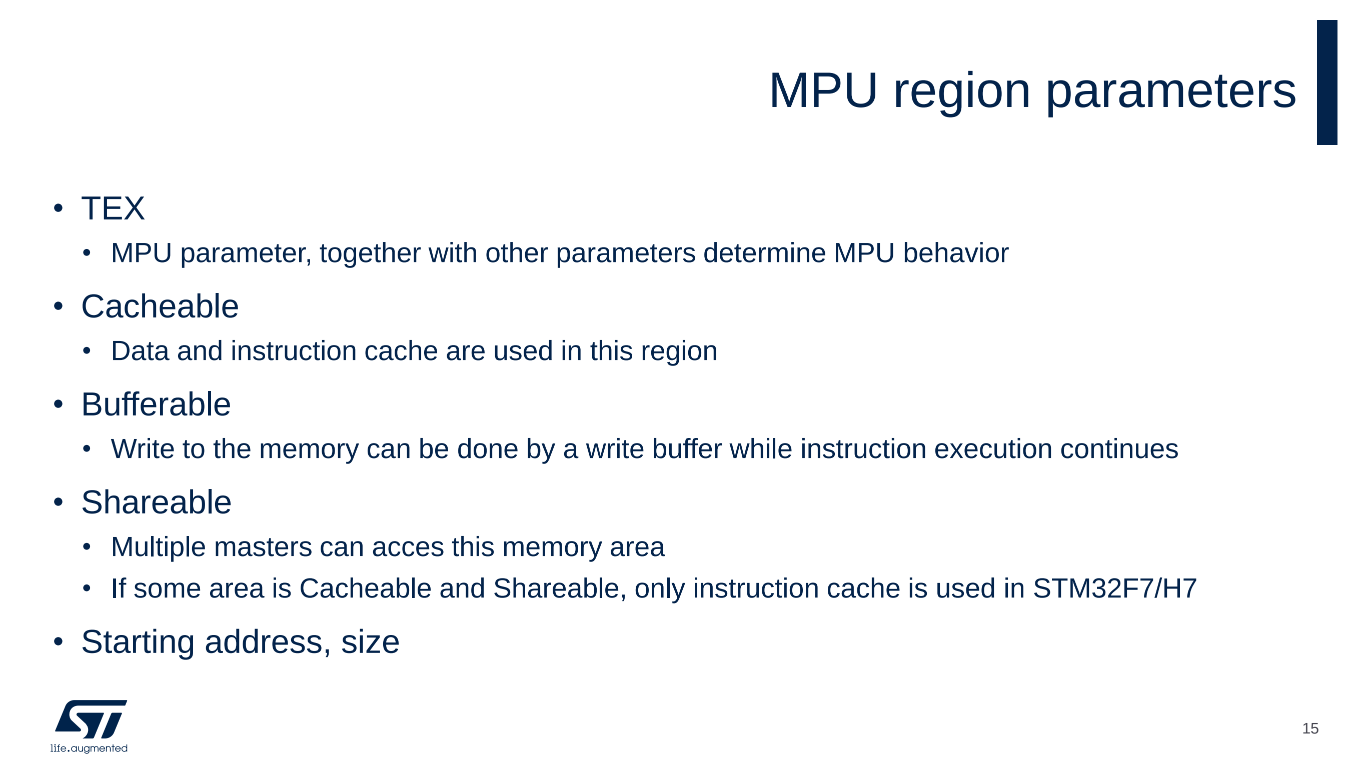
life.augmented

By enabling MPU in a project based on Cube HAL libraries, you have additionally one parameter to pass into HAL MPU enable function.

There are four possible settings, which covers all combinations of settings to bits in MPU control registers:

- First bit sets if MPU is enabled during hard fault, NMI and fault mask handlers. If kept at zero, MPU is disabled during hard fault, NMI and fault mask handlers.
- Second bit: allow default memory map. If this bit is enabled, default memory map is used as a background region for privileged software accesses. In this case, background region acts as if it has region number -1. Any region that is defined and enabled has priority over this default map. If disabled, default memory map is disabled and any memory access to a location not covered by any enabled region causes a fault.

In examples which you will find later in this presentation, MPU_PRIVILEGED_DEFAULT parameter is used, meaning that MPU is disabled in fault and default memory map is enabled.



MPU region parameters

- TEX
  - MPU parameter, together with other parameters determine MPU behavior
- Cacheable
  - Data and instruction cache are used in this region
- Bufferable
  - Write to the memory can be done by a write buffer while instruction execution continues
- Shareable
  - Multiple masters can acces this memory area
  - If some area is Cacheable and Shareable, only instruction cache is used in STM32F7/H7
- Starting address, size

life.augmented

Few parameters need to be set also for each MPU region which will be used:

- its starting address of the region and
- its size,
- then TEX parameter, which together with other MPU region parameters determine region memory type and cache behavior.

Then you need to set parameters we already discussed on previous slides:
By enabling or disabling you can make region

- Cacheable,
- Bufferable and
- Shareable.

Not all combination of MPU regions parameters are allowed.
List of possible combinations is on slide 17. Not listed on this slide is

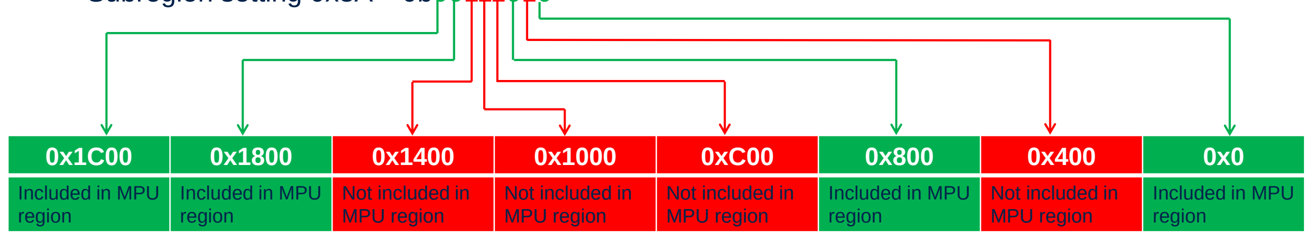
- parameter Execute Never.

This parameter can be set for any region without influence on use memory type.

Setting Execute Never on a region disables code execution from that region.

MPU subregions

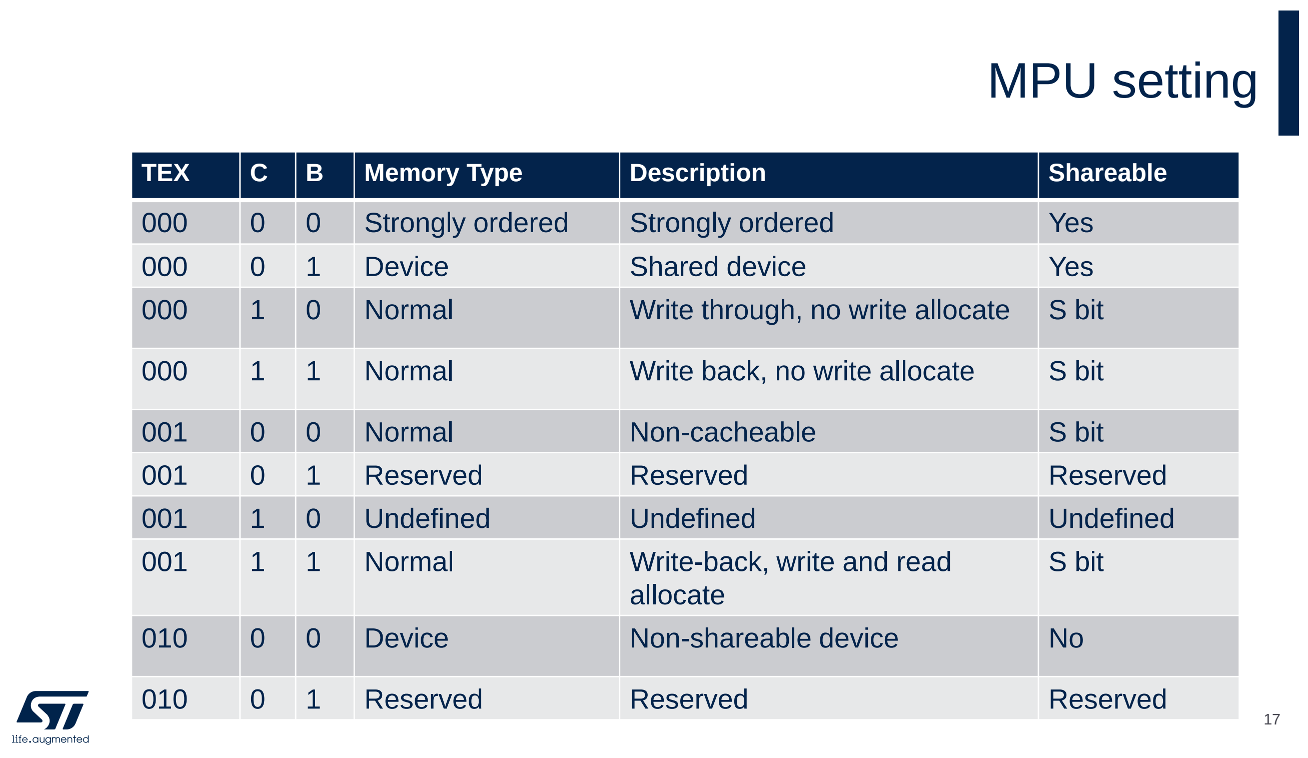
- Regions of 256 bytes or more are divided into 8 equal-sized subregions
  - For smaller regions subregions setting must be kept 0x0
- Setting corresponding subregion bit excludes subregion from MPU region
  - Settings in given MPU region are not applied on the subregion
- Example – 8 kB region, starting @ 0x0, each subregion size is 1 kB
  - Subregion setting 0x3A – 0b00111010



Last parameter in MPU region deserves more explanation. It's MPU sub-region setting.

For each region which size is 256 bytes or more, it is possible to divide region into 8 sub-regions with equal size. Excluding sub-region from region rules is set by writing one on corresponding position of 8-bit value.

Concrete usage is demonstrated on a picture.
We choose to have 8 kB region starting at address 0.
So each sub region size is 1 kB , and if sub region field is set to value 3A hex, the second, fourth, fifth and sixth sub-regions from start won't be included in MPU region.



MPU setting

TEX	C	B	Memory Type	Description	Shareable
000	0	0	Strongly ordered	Strongly ordered	Yes
000	0	1	Device	Shared device	Yes
000	1	0	Normal	Write through, no write allocate	S bit
000	1	1	Normal	Write back, no write allocate	S bit
001	0	0	Normal	Non-cacheable	S bit
001	0	1	Reserved	Reserved	Reserved
001	1	0	Undefined	Undefined	Undefined
001	1	1	Normal	Write-back, write and read allocate	S bit
010	0	0	Device	Non-shareable device	No
010	0	1	Reserved	Reserved	Reserved

life.augmented

Table on this slide lists all allowed MPU configuration in STM32 microcontrollers with memory protection unit.

Values from this table shall be followed when you design MPU for any region.

MPU cache policy

- Write through with no write allocate
  - On hits it writes to the cache and the main memory
  - On misses it updates the block in the main memory not bringing that block to the cache
- Write-back with no write allocate
  - On hits it writes to the cache setting dirty bit for the block, the main memory is not updated
  - On misses it updates the block in the main memory not bringing that block to the cache
- Write-back with write and read allocate
  - On hits it writes to the cache setting dirty bit for the block, the main memory is not updated
  - On misses it updates the block in the main memory and brings the block to the cache

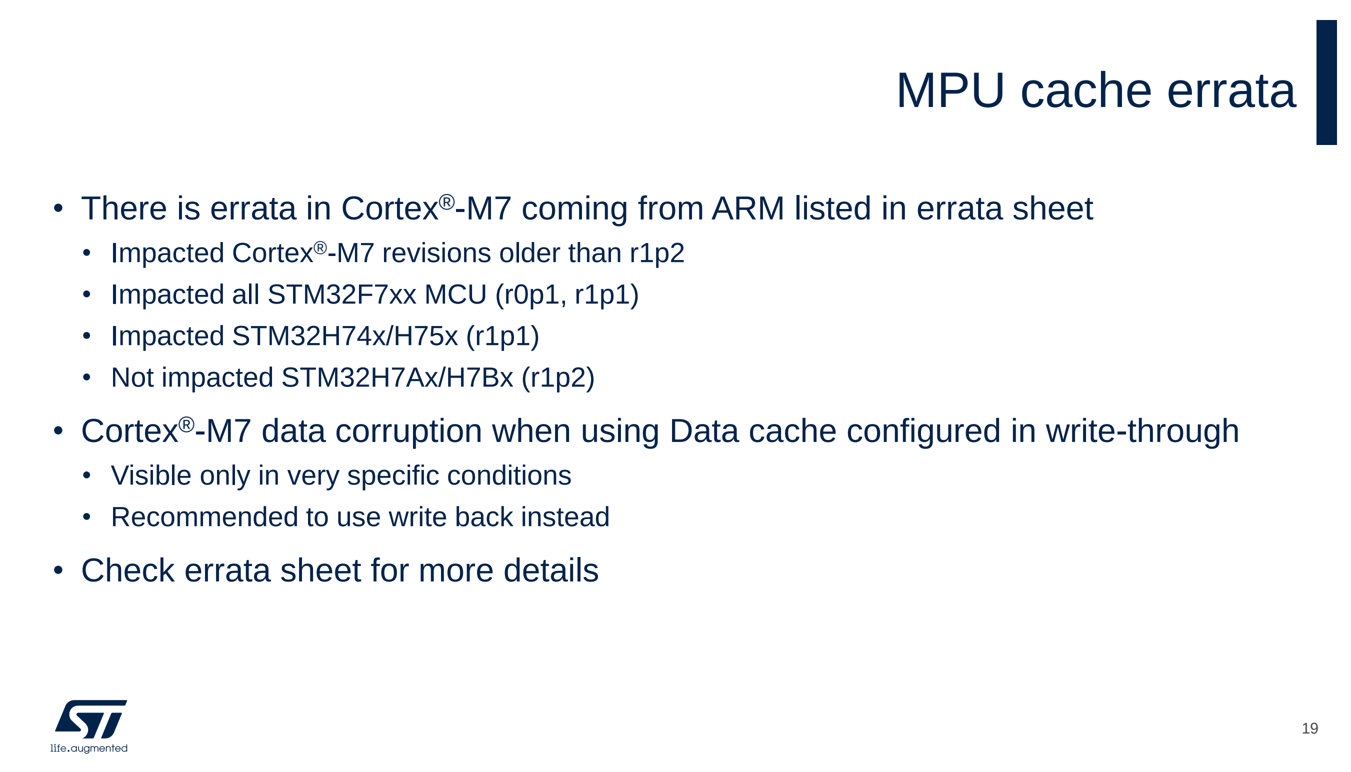
life.augmented

Settings in an MPU region also determines Cache Policy, when parameter Cacheable allowed in a region.

Cache Policy may have influence on performance based on memory kind and usage, different policy may be more efficient.

But it's not purpose of this material to cover more deeply Cache Policy setting.

For more information please find related materials at the end of this presentation.

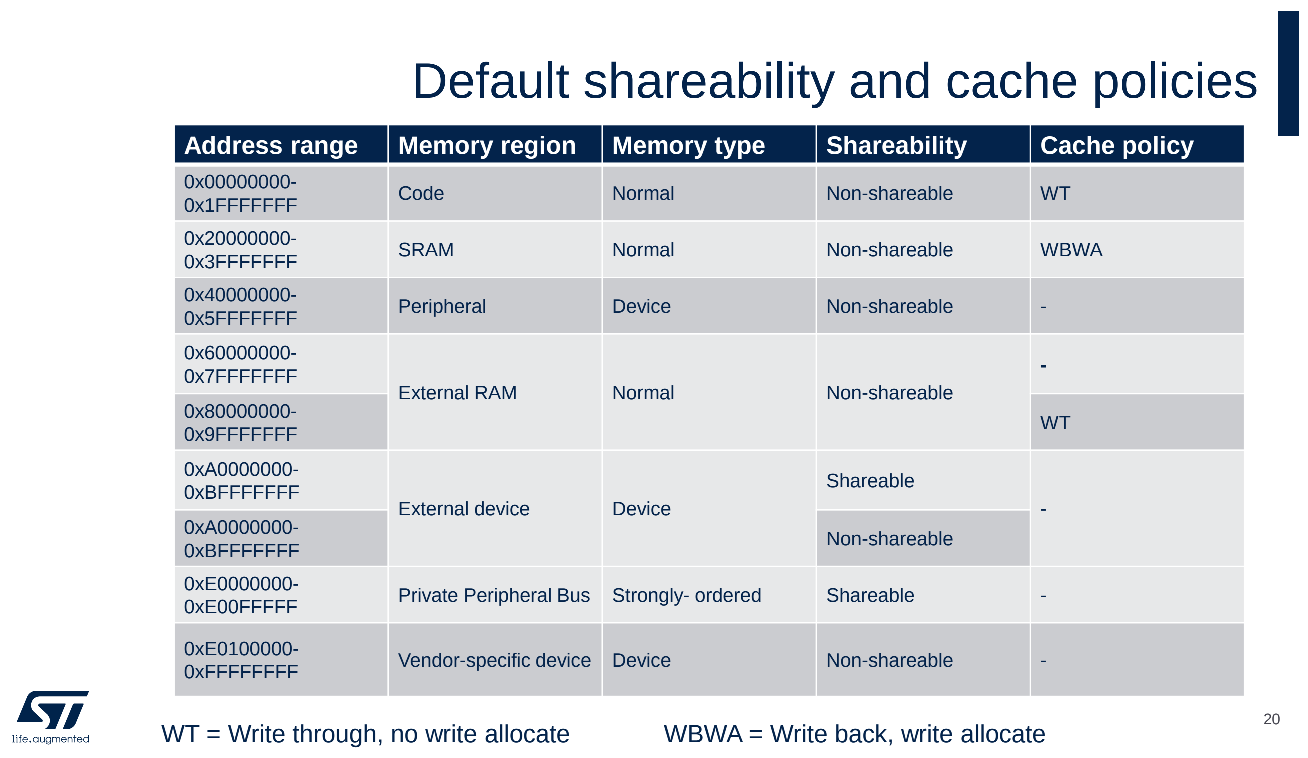


MPU cache errata

- There is errata in Cortex®-M7 coming from ARM listed in errata sheet
  - Impacted Cortex®-M7 revisions older than r1p2
  - Impacted all STM32F7xx MCU (r0p1, r1p1)
  - Impacted STM32H74x/H75x (r1p1)
  - Not impacted STM32H7Ax/H7Bx (r1p2)
- Cortex®-M7 data corruption when using Data cache configured in write-through
  - Visible only in very specific conditions
  - Recommended to use write back instead
- Check errata sheet for more details

Just would be good to warn you additionally, that in **STM 32F7 microcontrollers and some older STM32H7 microcontrollers, older revision of ARM Cortex-M7 is used, and this older version of Cortex-M7 has Errata for data cache usage when configured with write-through policy.**

Conditions to reproduce the issue are very specific. But to be safe, it's recommended to **use write back-policy instead**. For more details, please check product Errata sheet.



Default shareability and cache policies

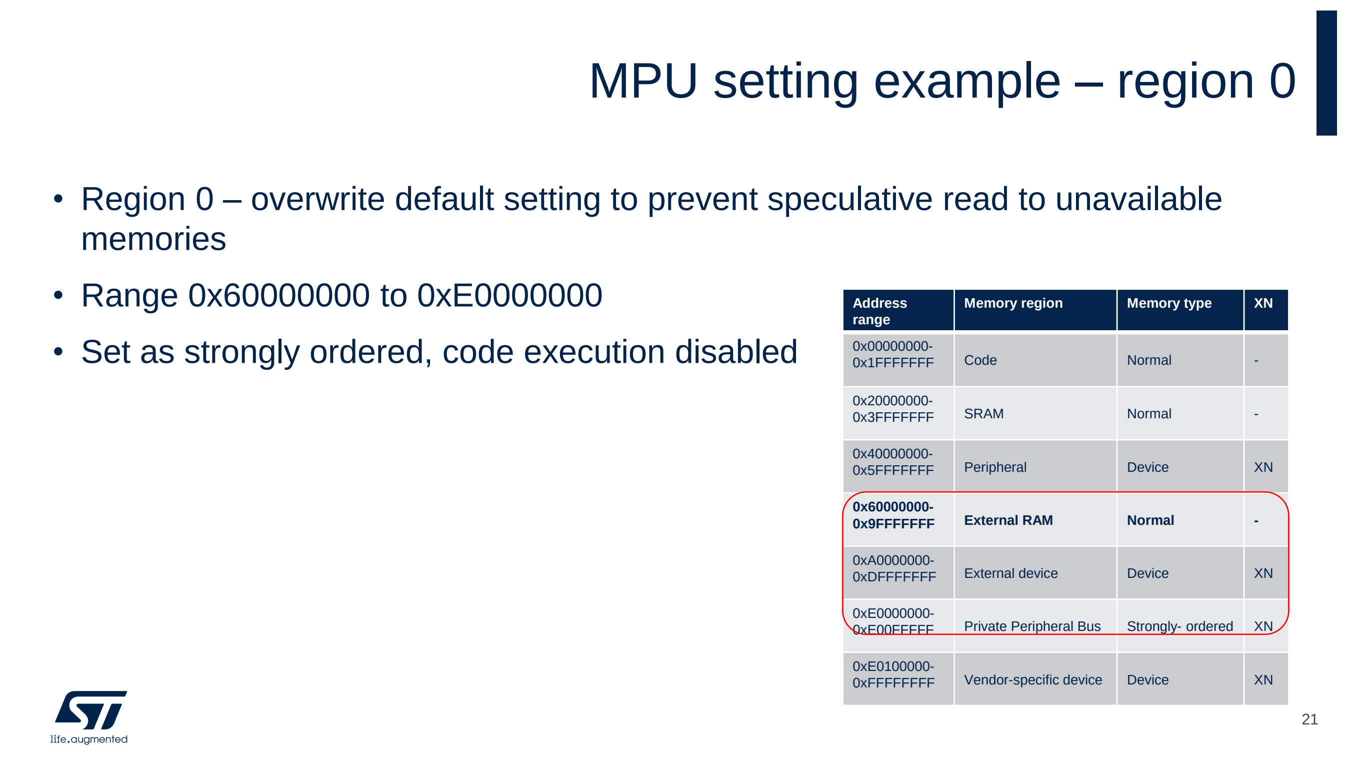
Address range	Memory region	Memory type	Shareability	Cache policy
0x00000000-0x1FFFFFFF	Code	Normal	Non-shareable	WT
0x20000000-0x3FFFFFFF	SRAM	Normal	Non-shareable	WBWA
0x40000000-0x5FFFFFFF	Peripheral	Device	Non-shareable	-
0x60000000-0x7FFFFFFF	External RAM	Normal	Non-shareable	-
0x80000000-0x9FFFFFFF				WT
0xA0000000-0xBFFFFFFF	External device	Device	Shareable	-
0xA0000000-0xBFFFFFFF			Non-shareable	
0xE0000000-0xE00FFFFF	Private Peripheral Bus	Strongly- ordered	Shareable	-
0xE0100000-0xFFFFFFFF	Vendor-specific device	Device	Non-shareable	-

WT = Write through, no write allocate WBWA = Write back, write allocate

Default setting of shareability and Cache Policies is visible in table.

By default, all cacheable regions are non-shareable. Software needs to handle data coherency there.

Also be careful that write-through Cache Policy is used on code memory region and part of external RAM memory region. Again, if this default setting is not suitable for your usage, you may change it using memory protection unit.



MPU setting example – region 0

- Region 0 – overwrite default setting to prevent speculative read to unavailable memories
- Range 0x60000000 to 0xE0000000
- Set as strongly ordered, code execution disabled

Address range	Memory region	Memory type	XN
0x00000000-0x1FFFFFFF	Code	Normal	-
0x20000000-0x3FFFFFFF	SRAM	Normal	-
0x40000000-0x5FFFFFFF	Peripheral	Device	XN
0x60000000-0x9FFFFFFF	**External RAM**	**Normal**	**-**
0xA0000000-0xDFFFFFFF	External device	Device	XN
0xE0000000-0xE00FFFFF	Private Peripheral Bus	Strongly- ordered	XN
0xE0100000-0xFFFFFFFF	Vendor-specific device	Device	XN

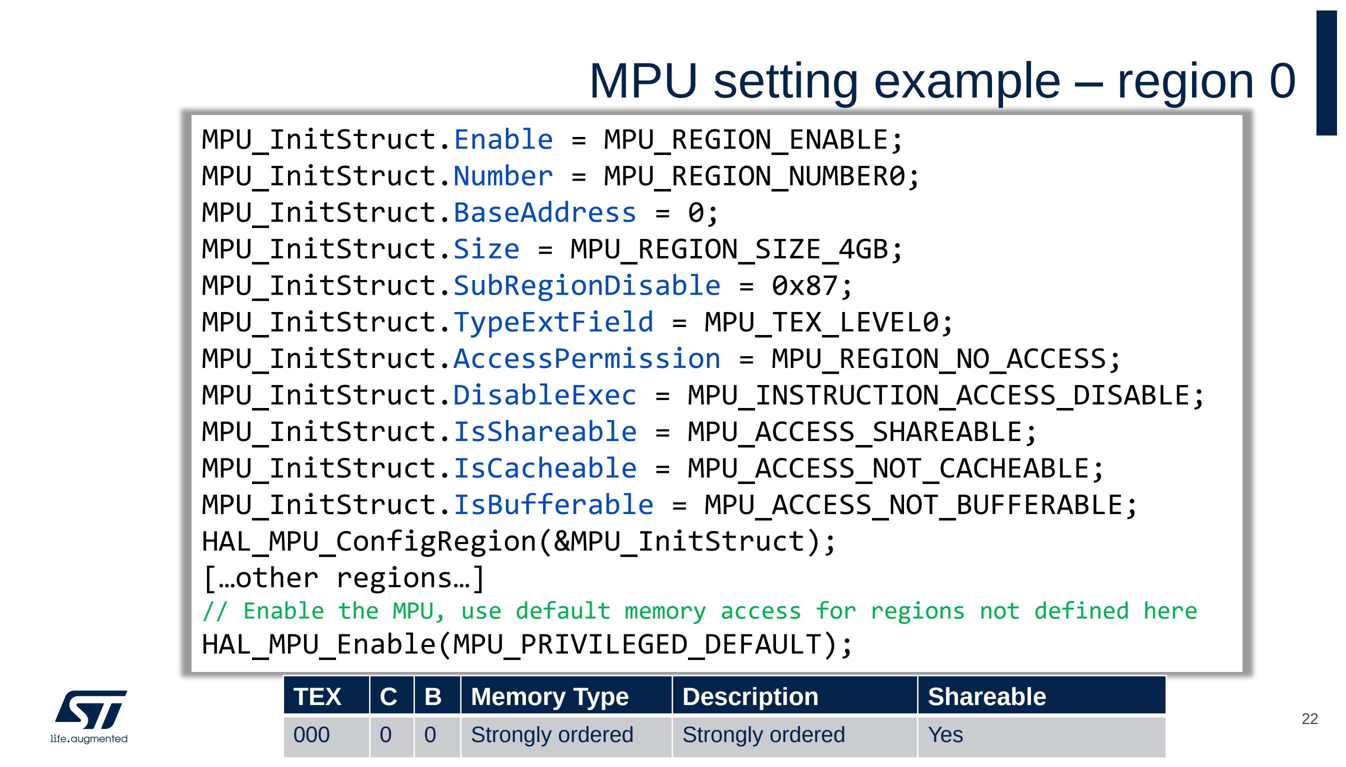
life.augmented

Now we're leaving the theoretical part and moving to examples of MPU settings, starting with region 0 which we recommend to set as basic region.

The purpose of the [MPU] region is to set [address] regions of unused memory - which can cause speculative access issue - into safe configuration which is not allowing speculative access.

Region is set as strongly ordered with Execute Never parameter set using sub-region.

Range 60 000 000 hex to E0 000 000 hex is set in region 0.



MPU setting example – region 0

```
MPU_InitStruct.Enable = MPU_REGION_ENABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBER0;
MPU_InitStruct.BaseAddress = 0;
MPU_InitStruct.Size = MPU_REGION_SIZE_4GB;
MPU_InitStruct.SubRegionDisable = 0x87;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.AccessPermission = MPU_REGION_NO_ACCESS;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_DISABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
HAL_MPU_ConfigRegion(&MPU_InitStruct);
[…other regions…]
// Enable the MPU, use default memory access for regions not defined here
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
```

TEX	C	B	Memory Type	Description	Shareable
000	0	0	Strongly ordered	Strongly ordered	Yes

life.augmented

Code example: how region 0 for preventing speculative memory read can be implemented in project (using sub-region usage address 60 000 000 hex to E0 000 000 hex is covered).

MPU setting example

- Usage of other regions strongly depends on the application and used interfaces
- Upcoming examples expect usage of region 0 as on previous slide
  - External QSPI flash memory
  - External SDRAM on FMC
  - DMA usage – RAM buffer
  - Ethernet – DMA RAM buffers
  - LCD display

A few more typical use cases of MPU usage are covered in this presentation.

There is example for

- setting external Q-SPI flash memory,
- SDRAM on FMC,
- DMA usage with internal RAM,
- RAM buffers for Ethernet, and
- LCD configuration.

All examples count with usage of region 0 from previous slide to prevent speculative read issue.

General region setting

- Device and Strongly ordered memory types don’t allows unaligned memory access
  - Normal memory type shall be used areas used for code execution
- For NAND memories and FPGAs each write should be executed sequentially
  - Usually connected using FMC interface
  - Don’t allow bufferable
  - Strongly ordered is best option
- MCU registers – device memory type

life.augmented

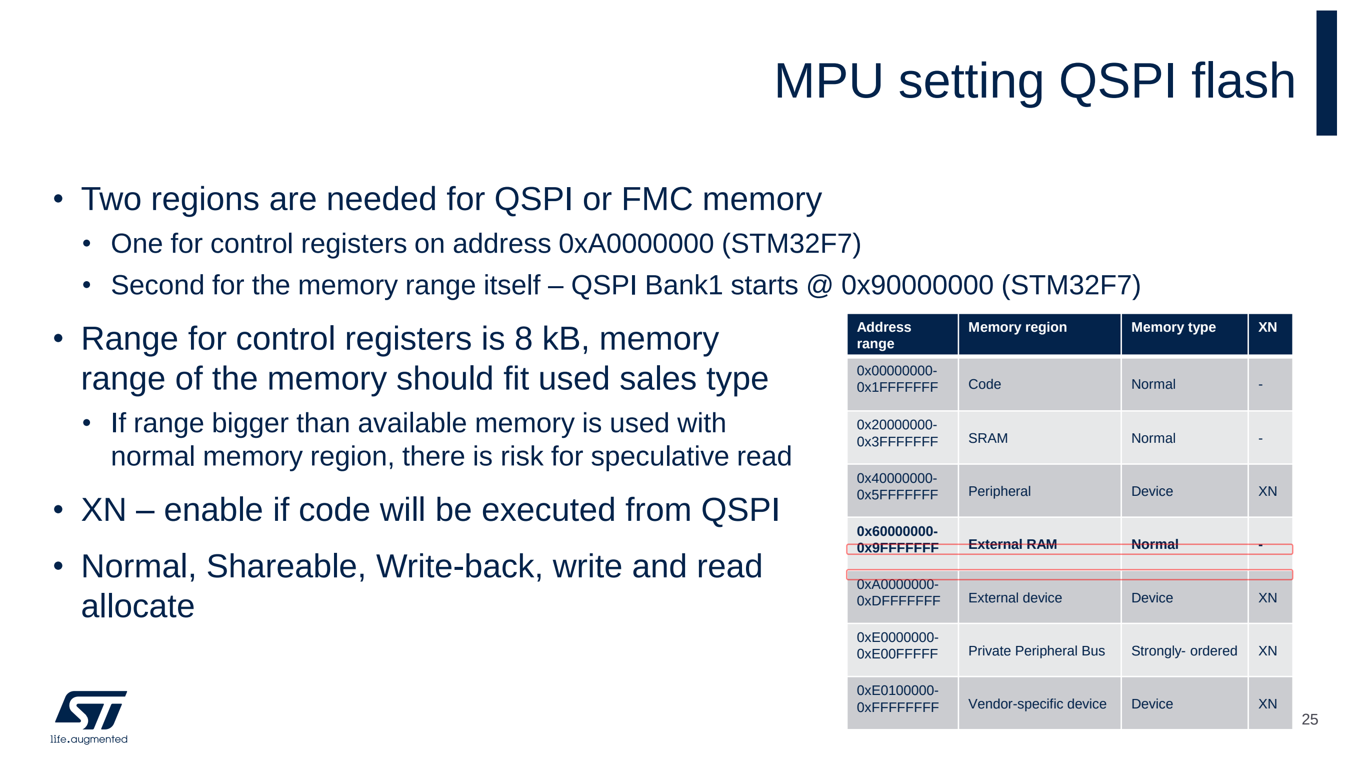
As already covered on start of this presentation, there are some general recommendations which memory types shall be used for various kinds of memories.

For **code execution** it’s best option to use **Normal memory type,** which allows unaligned memory access.

Also for **RAM** memories is **Normal memory type** convenient as it's not having additional restriction and offers best performance.

For **MCU registers,** access is important to preserve program order of instruction. instruction can be written in a burst using buffer will attribute so **Device** memory type is best option.

Strongly Ordered memory type is used in **memories which need to have each write be a single transaction -** for example NAND memories or FPGAs.



MPU setting QSPI flash

- Two regions are needed for QSPI or FMC memory
  - One for control registers on address 0xA0000000 (STM32F7)
  - Second for the memory range itself – QSPI Bank1 starts @ 0x90000000 (STM32F7)
- Range for control registers is 8 kB, memory range of the memory should fit used sales type
  - If range bigger than available memory is used with normal memory region, there is risk for speculative read
- XN – enable if code will be executed from QSPI
- Normal, Shareable, Write-back, write and read allocate

Address range	Memory region	Memory type	XN
0x00000000-0x1FFFFFFF	Code	Normal	-
0x20000000-0x3FFFFFFF	SRAM	Normal	-
0x40000000-0x5FFFFFFF	Peripheral	Device	XN
0x60000000-0x9FFFFFFF	**External RAM**	**Normal**	**-**
0xA0000000-0xDFFFFFFF	External device	Device	XN
0xE0000000-0xE00FFFFF	Private Peripheral Bus	Strongly- ordered	XN
0xE0100000-0xFFFFFFFF	Vendor-specific device	Device	XN

life.augmented

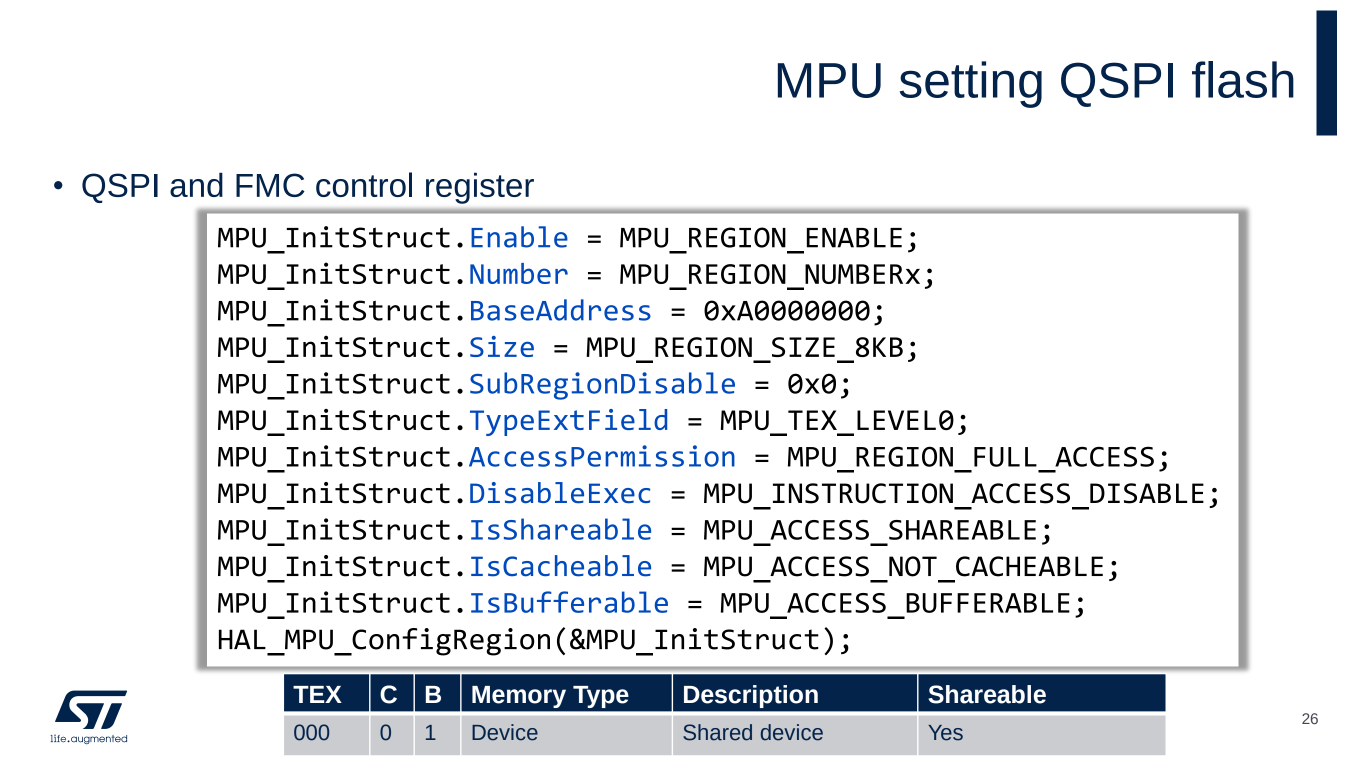
First example is external Q-SPI flash memory, those days often used as memory for code execution or large data storage.

In STM32F7 family, Q-SPI or FMC need to rewrite setting of region 0 to allow access on address a 0 000 000 hex, where Q-SPI and FMC control registers are located.

STM32H7 microcontrollers use a different address for Q-SPI and FMC control register storage, only one MPU region for Q-SPI memory ranges enough.

The Start of Q-SPI memory depends on used bank
Usually its address is 90 000 000 hex.

We recommend to set this region as Normal, Shareable with Cache write-back policy. If code won't be executed from the memory, also set Execute Never attribute.



MPU setting QSPI flash

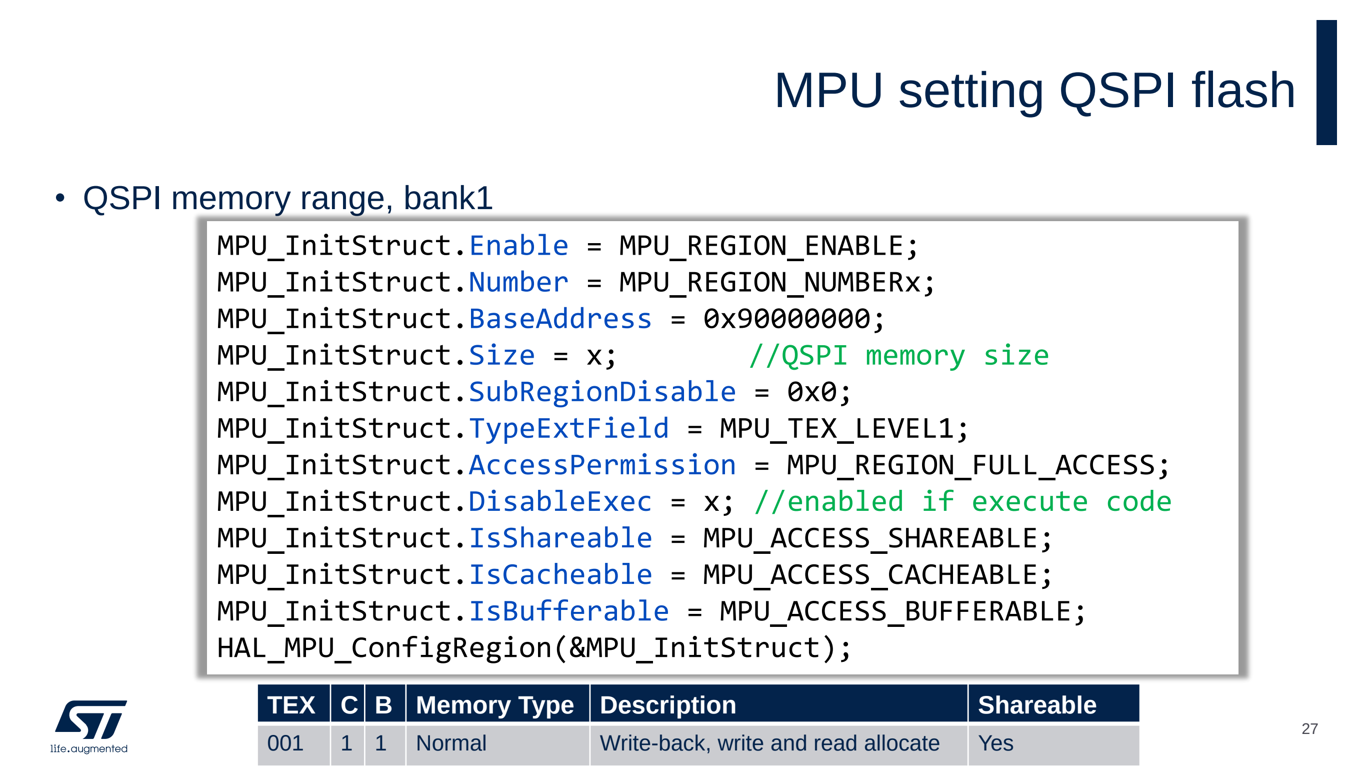
- QSPI and FMC control register

```
MPU_InitStruct.Enable = MPU_REGION_ENABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBERx;
MPU_InitStruct.BaseAddress = 0xA0000000;
MPU_InitStruct.Size = MPU_REGION_SIZE_8KB;
MPU_InitStruct.SubRegionDisable = 0x0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_DISABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

TEX	C	B	Memory Type	Description	Shareable
000	0	1	Device	Shared device	Yes

life.augmented

Code example setting Q-SPI and FMC control register access,



MPU setting QSPI flash

- QSPI memory range, bank1

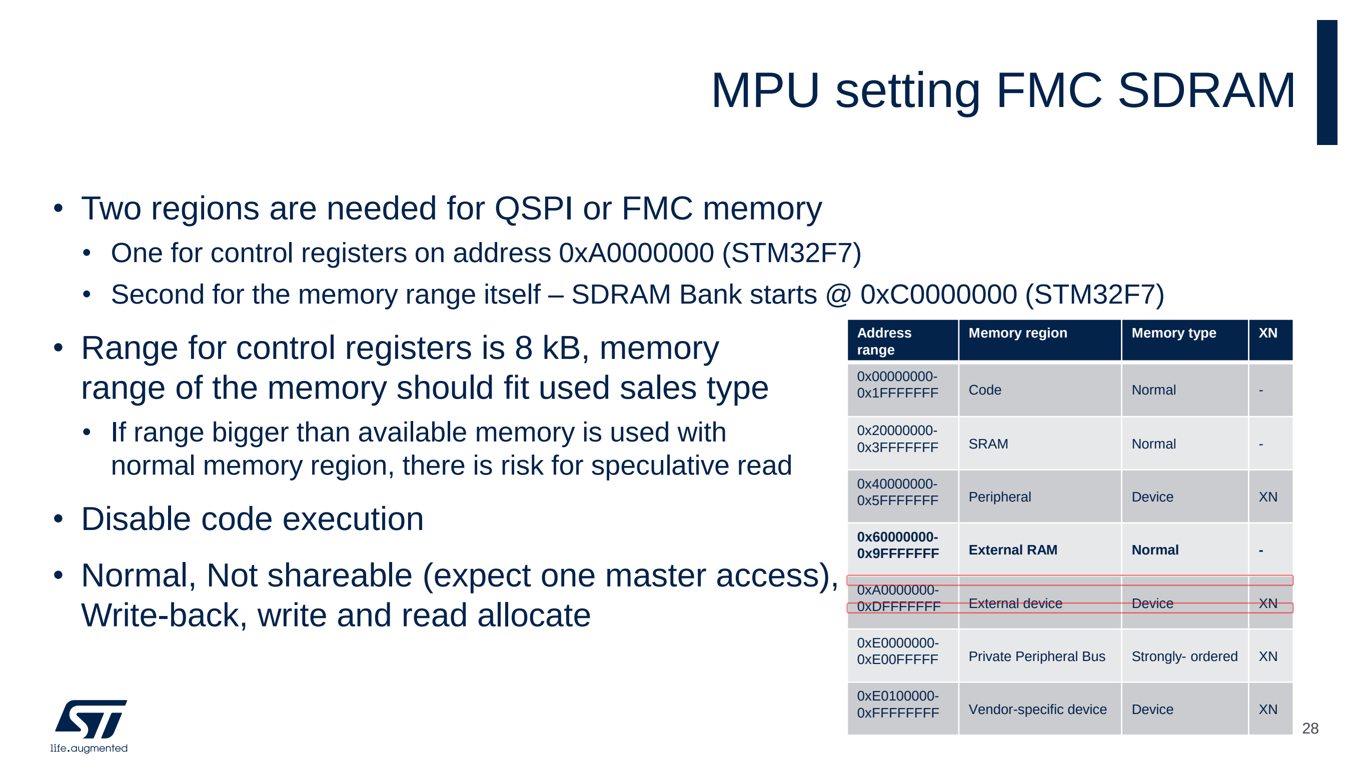
```
MPU_InitStruct.Enable = MPU_REGION_ENABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBERx;
MPU_InitStruct.BaseAddress = 0x90000000;
MPU_InitStruct.Size = x;        //QSPI memory size
MPU_InitStruct.SubRegionDisable = 0x0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.DisableExec = x; //enabled if execute code
MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

TEX	C	B	Memory Type	Description	Shareable
001	1	1	Normal	Write-back, write and read allocate	Yes

life.augmented

27

and initialization code for Q-SPI memory range.



MPU setting FMC SDRAM

- Two regions are needed for QSPI or FMC memory
  - One for control registers on address 0xA0000000 (STM32F7)
  - Second for the memory range itself – SDRAM Bank starts @ 0xC0000000 (STM32F7)
- Range for control registers is 8 kB, memory range of the memory should fit used sales type
  - If range bigger than available memory is used with normal memory region, there is risk for speculative read
- Disable code execution
- Normal, Not shareable (expect one master access), Write-back, write and read allocate

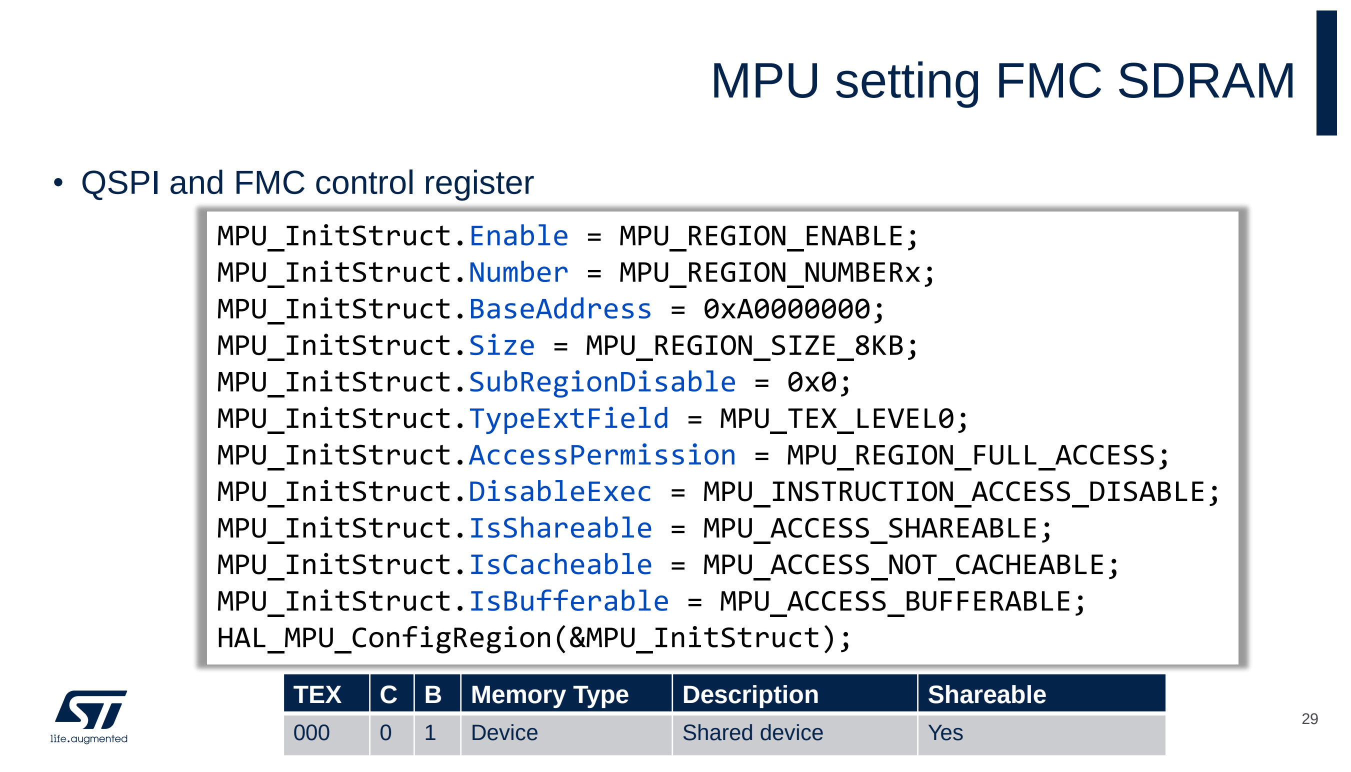
Address range	Memory region	Memory type	XN
0x00000000-0x1FFFFFFF	Code	Normal	-
0x20000000-0x3FFFFFFF	SRAM	Normal	-
0x40000000-0x5FFFFFFF	Peripheral	Device	XN
0x60000000-0x9FFFFFFF	**External RAM**	**Normal**	**-**
0xA0000000-0xDFFFFFFF	External device	Device	XN
0xE0000000-0xE00FFFFF	Private Peripheral Bus	Strongly- ordered	XN
0xE0100000-0xFFFFFFFF	Vendor-specific device	Device	XN

life.augmented

Like for Q-SPI, FMC on STM32F7 also needs a region added to allow access to control registers starting on address a 0 000 000 hex.

Start of SDRAM depends on used bank. Usually SDRAM is mapped on bank 1 at address C0 000 000 hex.

We recommend to set as **Normal** memory type with **write-back** Cache Policy.
If only one bus master is accessing the memory area, set as NOT_SHAREABLE to use also data cache.



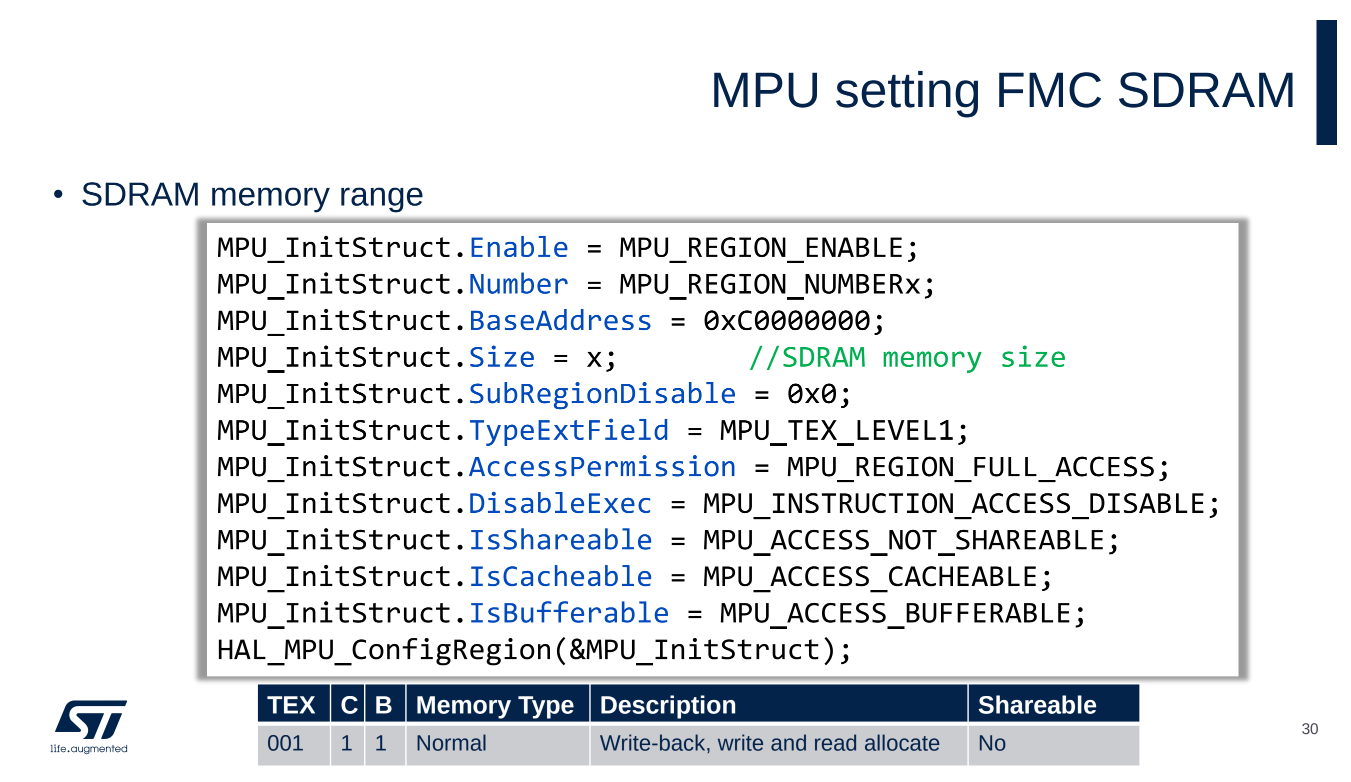
MPU setting FMC SDRAM

- QSPI and FMC control register

```
MPU_InitStruct.Enable = MPU_REGION_ENABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBERx;
MPU_InitStruct.BaseAddress = 0xA0000000;
MPU_InitStruct.Size = MPU_REGION_SIZE_8KB;
MPU_InitStruct.SubRegionDisable = 0x0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_DISABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

TEX	C	B	Memory Type	Description	Shareable
000	0	1	Device	Shared device	Yes

Code example setting Q-SPI and FMC control register access,



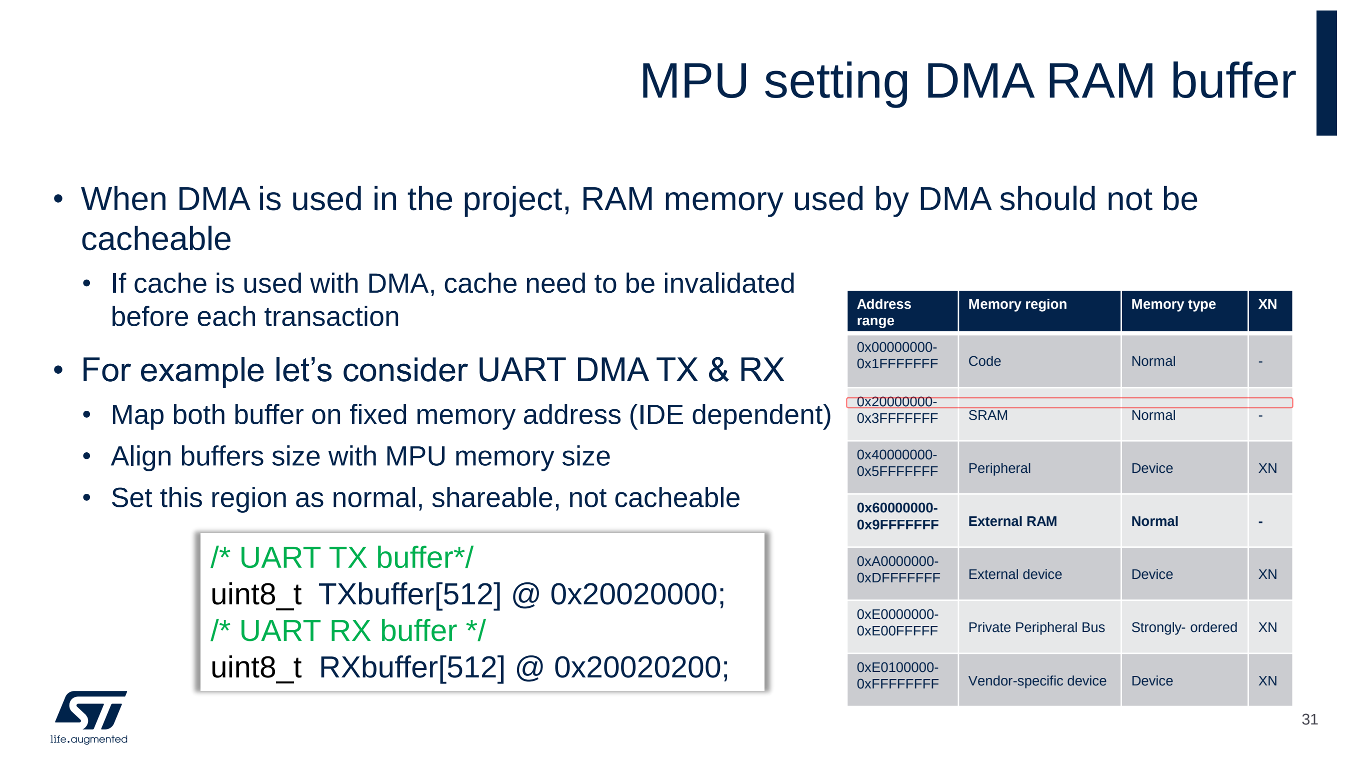
MPU setting FMC SDRAM

- SDRAM memory range

```
MPU_InitStruct.Enable = MPU_REGION_ENABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBERx;
MPU_InitStruct.BaseAddress = 0xC0000000;
MPU_InitStruct.Size = x;        //SDRAM memory size
MPU_InitStruct.SubRegionDisable = 0x0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_DISABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

TEX	C	B	Memory Type	Description	Shareable
001	1	1	Normal	Write-back, write and read allocate	No

and SDRAM memory range setting.



MPU setting DMA RAM buffer

- When DMA is used in the project, RAM memory used by DMA should not be cacheable
  - If cache is used with DMA, cache need to be invalidated before each transaction
- For example let’s consider UART DMA TX & RX
  - Map both buffer on fixed memory address (IDE dependent)
  - Align buffers size with MPU memory size
  - Set this region as normal, shareable, not cacheable

```
/* UART TX buffer*/
uint8_t  TXbuffer[512] @ 0x20020000;
/* UART RX buffer */
uint8_t  RXbuffer[512] @ 0x20020200;
```

Address range	Memory region	Memory type	XN
0x00000000-0x1FFFFFFF	Code	Normal	-
0x20000000-0x3FFFFFFF	SRAM	Normal	-
0x40000000-0x5FFFFFFF	Peripheral	Device	XN
0x60000000-0x9FFFFFFF	**External RAM**	**Normal**	**-**
0xA0000000-0xDFFFFFFF	External device	Device	XN
0xE0000000-0xE00FFFFF	Private Peripheral Bus	Strongly- ordered	XN
0xE0100000-0xFFFFFFFF	Vendor-specific device	Device	XN

life.augmented

31

DMA is very often used with microcontroller internal SRAM.
By default, RAM memory region is Shareable. That makes program responsible to keep data coherent when cache is used. **If you want to achieve optimal performance, it’s strongly recommended to enable both data and instruction cache.**

If you don't want to keep data coherency using software (which means flush complete cache before each DMA usage), best option is to **set just the part of RAM memory which is used by DMA as Shareable**.
That ensures data coherency of the memory and preserves data cache enabled for parts of RAM where application can safely use it, meaning DMA is not used there.

In an example here we choose two buffers in RAM, total size 1 kB.
In program, buffer address will be fixed to start from address 20 020 000 hex.

MPU setting DMA RAM buffer

```
MPU_InitStruct.Enable = MPU_REGION_ENABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBERx;
MPU_InitStruct.BaseAddress = 0x20020000;
MPU_InitStruct.Size = MPU_REGION_SIZE_1KB;
MPU_InitStruct.SubRegionDisable = 0x0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_DISABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

TEX	C	B	Memory Type	Description	Shareable
001	0	0	Normal	Non-cacheable	Yes

Code setting for MPU region:

This setting has multiple possibilities.

As instructions won't be saved in the region, it makes no difference if cache is completely disabled or cache is enabled when set as Shareable - then data cache is not used again.

MPU setting Ethernet

- Four buffers accessed by DMA are used, for better performance are those buffers linked into SRAM2 (STM32F7)

```
/* Ethernet Rx DMA Descriptors */
ETH_DMADescTypeDef  DMARxDscrTab[4] @ 0x2004C000;
/* Ethernet Tx DMA Descriptors */
ETH_DMADescTypeDef  DMATxDscrTab[4] @ 0x2004C080;
/* Ethernet Receive Buffers */
uint8_t Rx_Buff[4][1524] @ 0x2004C100;
/* Ethernet Transmit Buffers */
uint8_t Tx_Buff[4][1524] @ 0x2004D8D0;
```

life.augmented

Ethernet peripheral in STM32 uses DMA for data transfers between peripheral and buffers placed in RAM.

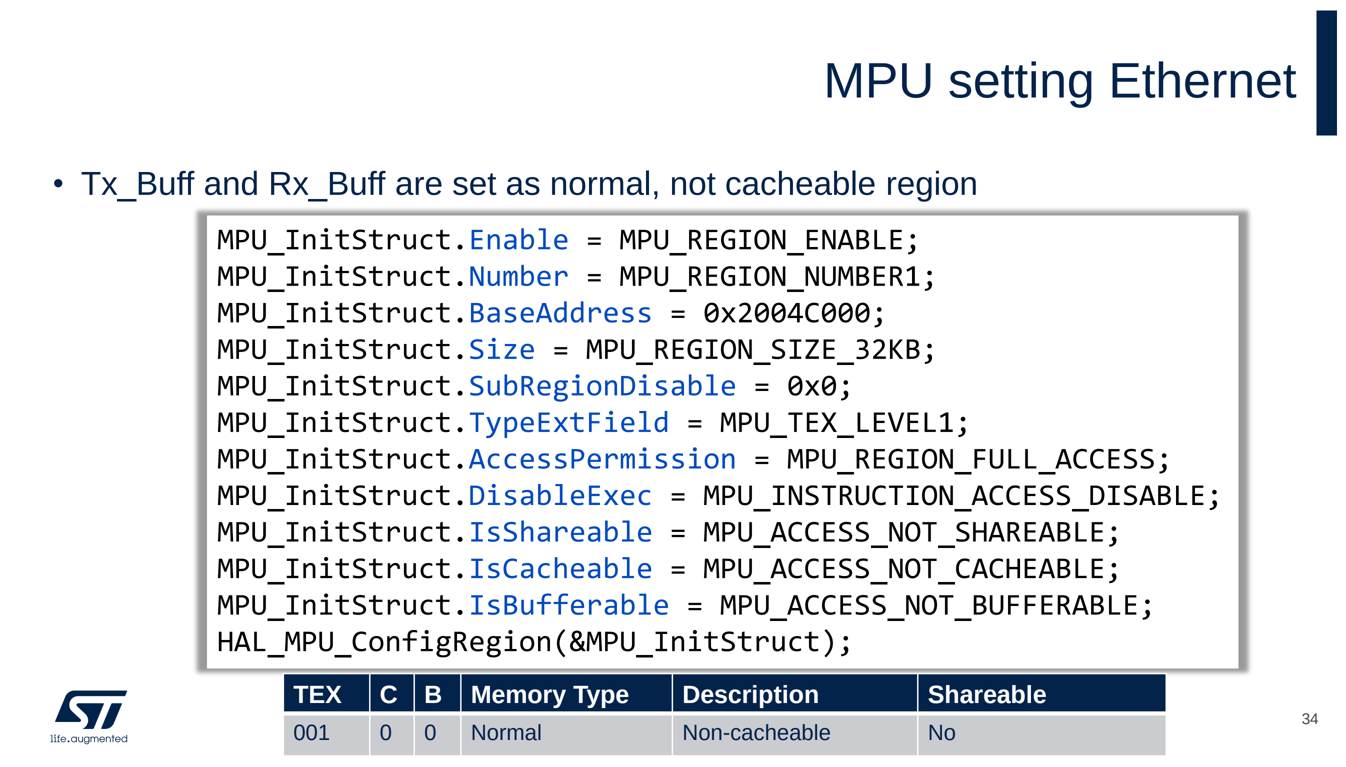
Like frame buffer used by DMA with cache usage discussed in previous part, **region where Ethernet RAM buffers are placed, need to be set as Shareable.**

Additionally, it's recommended to use different RAM for Ethernet buffer and application data to achieve better performance.

For example, on STM32F746, use SRAM1 for application data storage and SRAM2 for Ethernet buffers.

Ethernet peripheral demands two pairs of buffer:

- one for data itself,
- second for DMA descriptor tables.



MPU setting Ethernet

- Tx_Buff and Rx_Buff are set as normal, not cacheable region

```
MPU_InitStruct.Enable = MPU_REGION_ENABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBER1;
MPU_InitStruct.BaseAddress = 0x2004C000;
MPU_InitStruct.Size = MPU_REGION_SIZE_32KB;
MPU_InitStruct.SubRegionDisable = 0x0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_DISABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

TEX	C	B	Memory Type	Description	Shareable
001	0	0	Normal	Non-cacheable	No

Transmit and receive buffer are RAM storage with DMA usage.

Set as **Normal**, **NOT_CACHEABLE** memory region.
As cache is disabled, **Shareable don't need to be enabled**.

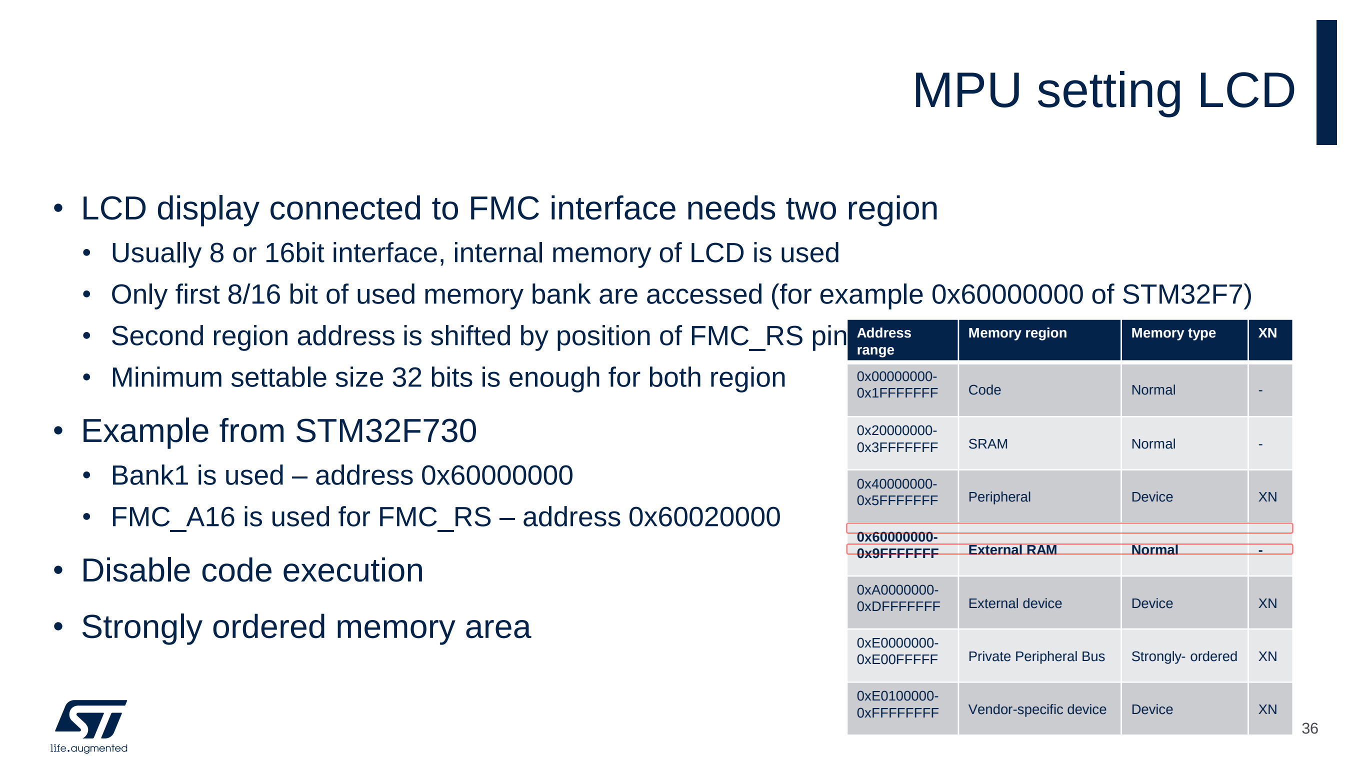
MPU setting Ethernet

- DMARxDscrTab and DMATxDscrTab get own subregion to set as shared device region

```
MPU_InitStruct.Enable = MPU_REGION_ENABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBER2;
MPU_InitStruct.BaseAddress = 0x2004C000;
MPU_InitStruct.Size = MPU_REGION_SIZE_256B;
MPU_InitStruct.SubRegionDisable = 0x0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_DISABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

TEX	C	B	Memory Type	Description	Shareable
000	0	1	Device	Shared device	Yes

DMA descriptor tables shall be set as Shared device memory.



MPU setting LCD

- LCD display connected to FMC interface needs two region
  - Usually 8 or 16bit interface, internal memory of LCD is used
  - Only first 8/16 bit of used memory bank are accessed (for example 0x60000000 of STM32F7)
  - Second region address is shifted by position of FMC_RS pin
  - Minimum settable size 32 bits is enough for both region
- Example from STM32F730
  - Bank1 is used – address 0x60000000
  - FMC_A16 is used for FMC_RS – address 0x60020000
- Disable code execution
- Strongly ordered memory area

Address range	Memory region	Memory type	XN
0x00000000-0x1FFFFFFF	Code	Normal	-
0x20000000-0x3FFFFFFF	SRAM	Normal	-
0x40000000-0x5FFFFFFF	Peripheral	Device	XN
0x60000000-0x9FFFFFFF	**External RAM**	**Normal**	**-**
0xA0000000-0xDFFFFFFF	External device	Device	XN
0xE0000000-0xE00FFFFF	Private Peripheral Bus	Strongly- ordered	XN
0xE0100000-0xFFFFFFFF	Vendor-specific device	Device	XN

life.augmented

Last example of MPU setting in this presentation is for LCD display controlled by flexible memory controller (FMC).

In such scenario, internal memory in LCD display is typically used, and 8 or 16 bit bus width is used.

Depending on FMC register select pin usage, two different memory regions may be used. which may need to use two memory region with same settings.

That is also the case in the example here.
Region 1 covers 32 bytes of address 60 000 000 hex, which is the starting address of bank 1 on STM 32F7.

32 Bytes is minimal MPU region size.

MPU setting LCD

- LCD connected on FMC

```
MPU_InitStruct.Enable = MPU_REGION_ENABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBERx;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size = MPU_REGION_SIZE_32B;
MPU_InitStruct.SubRegionDisable = 0x0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_DISABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

TEX	C	B	Memory Type	Description	Shareable
000	0	0	Strongly ordered	Strongly ordered	Yes

life.augmented

First region setting 32 bytes, **Strongly Ordered** memory starting from 60 000 000 hex.

MPU setting LCD

- LCD connected on FMC – active RS

```
MPU_InitStruct.Enable = MPU_REGION_ENABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBERx;
MPU_InitStruct.BaseAddress = 0x60020000;
MPU_InitStruct.Size = MPU_REGION_SIZE_32B;
MPU_InitStruct.SubRegionDisable = 0x0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_DISABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

TEX	C	B	Memory Type	Description	Shareable
000	0	0	Strongly ordered	Strongly ordered	Yes

life.augmented

Second region for toggling register select also 32 bytes, **Strongly Ordered** memory in our configuration starting from address 60 020 000 hex

Related materials

- PM0253 - STM32F7 Series and STM32H7 Series Cortex®-M7 processor programming manual
- AN4839 - Level 1 cache on STM32F7 Series and STM32H7 Series
- AN4838 - Managing memory protection unit (MPU) in STM32 MCUs
- AN4861 - LCD-TFT display controller (LTDC) on STM32 MCUs
  - Contains also description of MCU speculative access and MPU setting for graphical applications

life.augmented

Here you can find references to other material types of memory protection unit from ST Microelectronics:

- Programming Manual for Cortex-M7 microcontrollers,
- Application Note about level 1 cache,
- dedicated application note about memory protection unit,
- and for some users surprising the also application note about LTDC which contains quite detailed description of MPU usage and also contains mentioned about speculative read lock.

You can explore this material to get more complete information about MPU usage.

In the presentation, addresses of microcontroller peripherals are often mentioned. They need to be be changed from family to family and you can find peripheral addresses and complete memory mapping reference manual.

Thank you

life.augmented

Thank you for watching this presentation and wish you lots of success with STM32.