Protection configuration in XMC7000 family

About this document

Scope and purpose

This application note explains the functionality and how to configure the protection units for XMC7000 family MCU. This document serves as a guide to enhance system security based on different operations. It also explains the structure, access attributes, and some usage examples of each protection unit.

Associated part family

XMC7000 family of XMC™ industrial microcontrollers.

Table of contents

1 Introduction

This application note describes the protection units in XMC7000 family series MCU. The series includes Arm® Cortex® CPUs with Enhanced Secure Hardware Extension (eSHE), CAN FD, memory, and analog and digital peripheral functions in a single chip.

The XMC7100 and XMC7200 series MCUs have two Arm® Cortex®-M7-based CPUs (CM7) and a Cortex®-M0+ CPU.

Protection units are an important part for security system design, and enforce security based on different operations. A protection unit allows or restricts bus transfers on the bus infrastructure based on specific properties. A protection violation is caused by a mismatch between a bus transfer's address region and access attributes and the protection structures' address range and access attributes.

These series have three types of protection units:

- Memory Protection Unit (MPU),
- Shared Memory Protection Unit (SMPU),
- Peripheral Protection Unit (PPU).

Memory protection is provided by the MPU and SMPU, and protection for peripheral resources is provided by the PPU.

The MPU, SMPU, and PPU protection structure definition follows the Arm® definition (in terms of memory region and access attribute definition) to ensure a consistent software interface.

If security is required, the SMPU and possibly PPUs registers must be controlled by a secured CPU that enforces system-wide protection.

To understand the functionality described and terminology used in this application note, see the "Protection Unit" chapter of the architecture reference manual.

In addition, this application note describes example code with the Peripheral Driver Library (PDL). The code snippets in this application note are part of PDL. See References for the PDL.

This sample program is for XMC7200 series.

2 Protection units

2.1 Location of protection units

Figure 1 shows the locations of MPUs, SMPUs, and PPUs in the XMC7200 series.

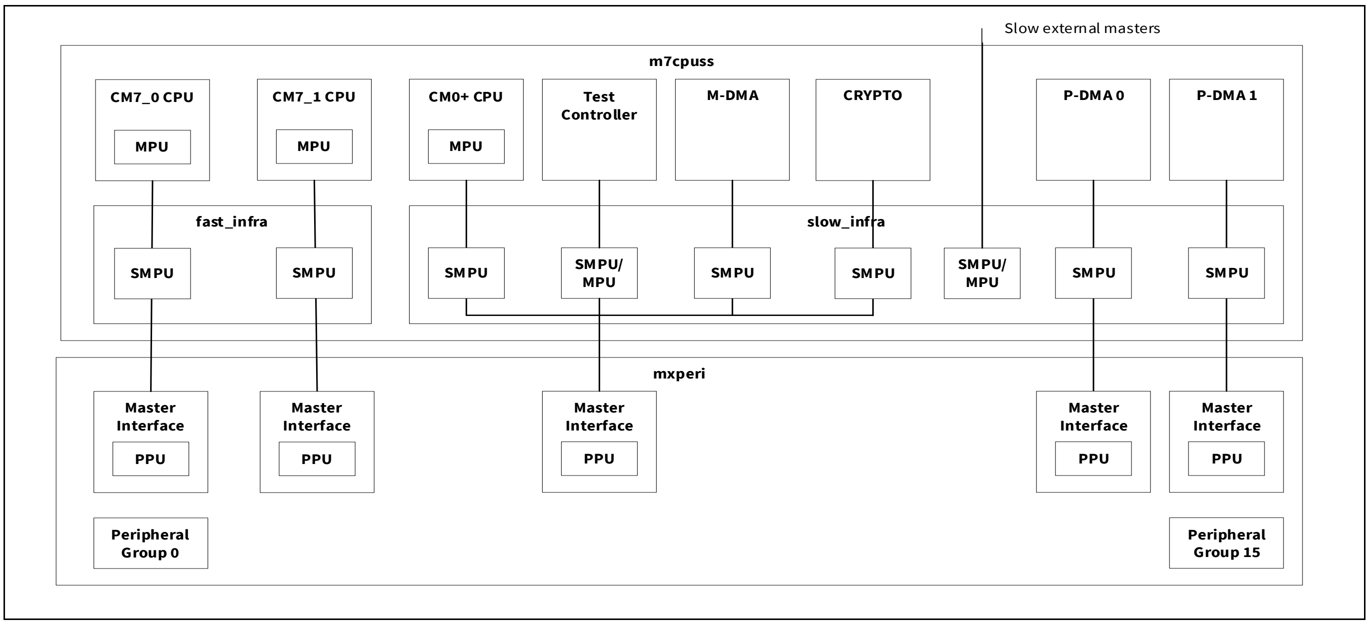

Figure 1 Protection unit locations in the XMC7200 series

See the architecture reference manual for the location of other series devices.

2.2 Protection units overview

MPUs are associated with a single master. There are following two types of MPUs:

- An MPU that is implemented as part of the CPU: This type is found in the Arm® CPUs
- An MPU that is implemented as part of the bus infrastructure: This type is found in bus masters such as test controller

However, Peripheral DMA (P-DMA/DW), Memory DMA (M-DMA/DMAC), and Cryptography (CRYPTO) components do not have an MPU. These masters inherit the access control attributes of the bus transfer that programmed channels or components.

An SMPU is shared by all bus masters. A single set of SMPU region structures provides the same protection information to all SMPUs in the systems.

A PPU is shared by all bus masters. The PPU provides access control to the peripherals within a peripheral group. PPUs are of two types:

- Fixed PPU: The address to protect is fixed and cannot be modified by software.
- Programmable PPU: The address to protect is programmable by software.

The MPU and SMPU have a higher priority over the PPU. In addition, a programmable PPU has a higher priority than a fixed PPU.

See the architecture reference manual or more details on protection units.

3 Operation overview

3.1 Protection properties of bus transfer

Protection units identify the following properties of bus transfer:

- An address range to be accessed

The MPU, SMPU, and PPU protection structure definition follows the Arm® definition. Therefore, note the following when setting the region address and region size of the protection unit.
- The base address of the region address must be aligned to the region size. When the region size is 64 KB, the base address must be aligned on a multiple of 64 KB, for example, at 0x00010000 or 0x00020000. See the registers reference manual for details.

- Access attributes such as the following:
  - Read/Write: Distinguish a read access from a write access
  - Execute: Distinguish a code access from a data access
  - User/Privileged: Distinguish a user access from a privileged access
  - Secured/non-secured: Distinguish a secured code access from a non-secured code access. The non-secured attribute allows both non-secured and secured accesses.
  - Protection context: Distinguish accesses from different protection contexts

Not all bus masters provide all these access attributes. No bus master has a protected context; Arm® CPUs do not have a secured attribute.

Access attributes not provided by the bus master are provided by the PROT_MPUx_MS_CTL and PROT_SMPU_MSx_CTL registers. These registers may be set during the boot process or by the secure CPU. Figure 2 shows the structure of PROT_MPUx_MS_CTL registers.

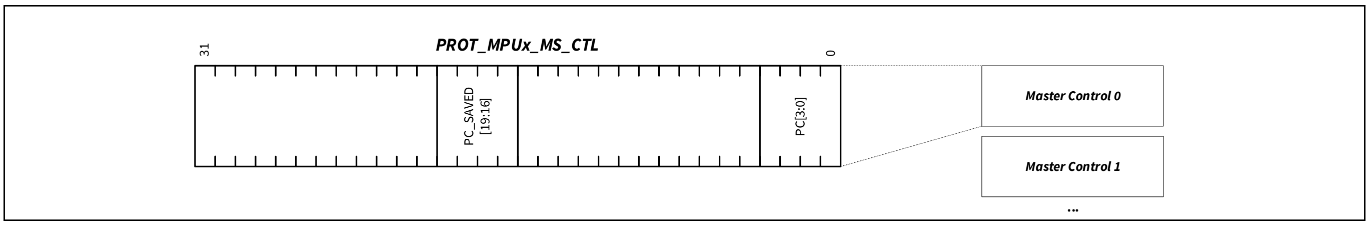

Figure 2 PROT_MPUx_MS_CTL register

This register grants a protection context attribute to its master access.

- PROT_MPUx_MS_CTL.PC: Sets the protection context attribute of its own access
- PROT_MPUx_MS_CTL.PC_SAVED: The boot process sets this field. This field is present only for the CM0+ master.

Figure 3 shows the structure of the PROT_SMPU_MSx_CTL registers.

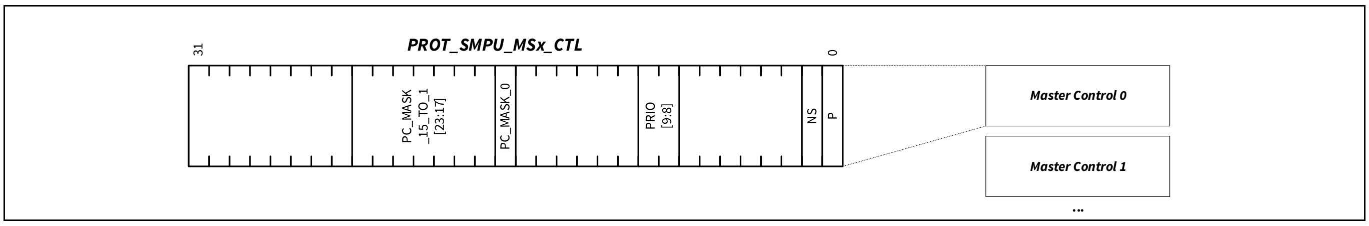

Figure 3 PROT_SMPU_MSx_CTL register

This register grants the following attributes to its master access:

- PROT_SMPU_MSx_CTL.P: Provides the User/Privileged attribute for masters that do not provide their own attributes.
- PROT_SMPU_MSx_CTL.NS: Provides the secured/non-secured attribute for masters that do not provide their own attributes.
- PROT_SMPU_MSx_CTL.PC_MASK_15_TO_1 and PC_MASK_0: Restricts the protection context that the bus master can set to MPUx_MS_CTL.PC.

Note: When one of the bits CPUSS_CM0_PC_CTL.VALID[3:1] is '1' (the associated protection context handler is valid), write transfers of the application software to the associated bits of PROT_SMPU_MSx_CTL.PC_MASK_15_TO_1[19:17] always writes '0'. This ensures that when valid protection context handlers are used to enter protection contexts 1, 2, or 3, the application software cannot enter those protection contexts. Also, the application software should clear the relevant bits of PROT_SMPU_MSx_CTL.PC_MASK_15_TO_1[19:17] when CPUSS_CMx_PC_CTL.VALID[3:1] bits are set to 1. See registers reference manual for more details.

- The PC_MASK_0 field is always ‘0’. This means that bus masters cannot set the PC = 0 attribute.
- PROT_SMPU_MSx_CTL.PRIO: Sets the bus arbitration priority.

However, not all bus masters have these register fields. Table 1 shows the relationship of registers for each master.

Table 1 Register field provided to the master

Register field	CM0+ CPU	CRYPTO component	P-DMA (DW)	M-DMA (DMAC)	CM7 CPU	Test controller
PROT_MPUx_MS_ CTL.PC	Yes	–	–	–	Yes	Yes
PROT_MPUx_MS_ CTL.PC_SAVED	Yes	–	–	–	–	–
PROT_SMPU_MSx_ CTL.P	–	–	–	–	–	Yes
PROT_SMPU_MSx_ CTL.NS	Yes	–	–	–	Yes	Yes
PROT_SMPU_MSx_ CTL.PC_MASK_15_TO_1 and PC_MASK_0	Yes	–	–	–	Yes	Yes
PROT_SMPU_MSx_ CTL.PRIO	Yes	Yes	Yes	Yes	Yes	Yes

P-DMA (DW), M-DMA (DMAC), and CRYPTO components do not have MPU. Therefore, these peripheral functions do not have fields to set attributes.

Each master has an associated SMPU MS_CTL register. However, in secure systems, this register can be typically controlled only by the secured master (CM0+) to prevent a master from changing its own privileged setting, security setting, arbitration priority, or enabled protection contexts.

3.2 Attribute inheritance

As mentioned earlier, P-DMA (DW), M-DMA (DMAC), and CRYPTO components inherit the access control attributes of the bus transfers that programmed the channels and components. The inherited access attribute is allowed/restricted by SMPU and PPU.

Figure 4 shows examples of the setting and behavior for inheriting the attributes.

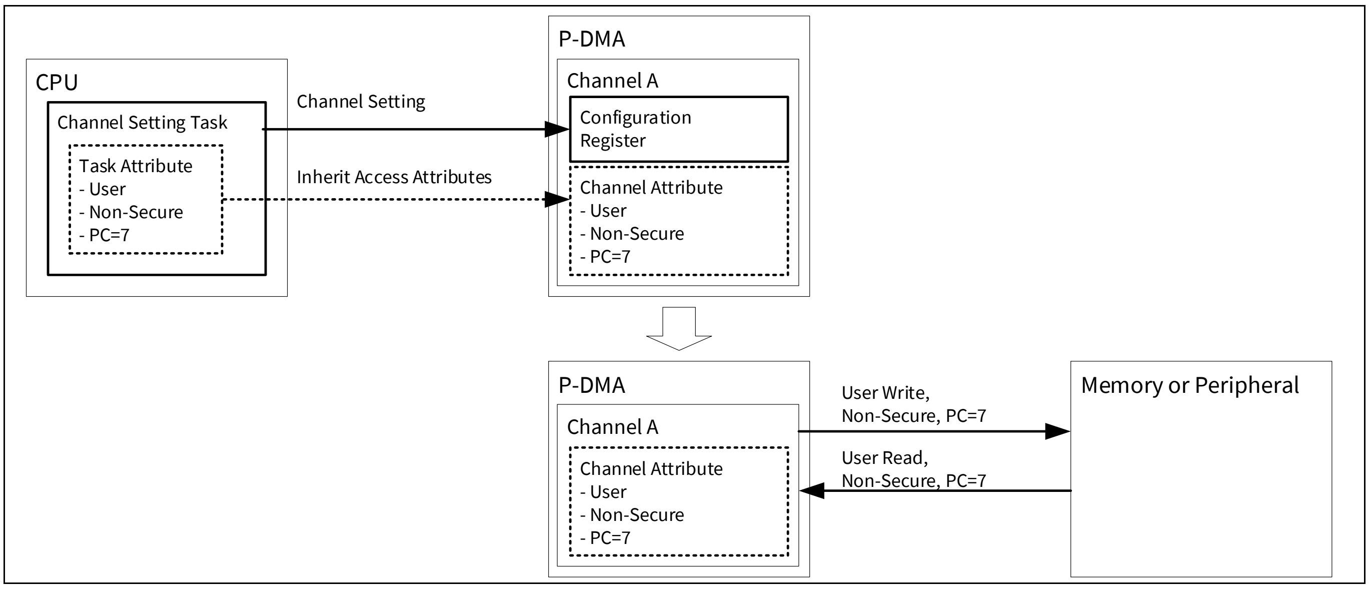

Figure 4 Setting and behavior example for attribute inheritance

3.3 User/Privileged attribute switching

This section describes attribute switching of both CPUs supporting User/Privileged attributes. CPUs support two operating modes and two privilege levels as follows:

- Operation mode
  - Thread mode: This mode is used to execute the application software. This mode can run in Privileged level or User level.
  - Handler mode: This mode is used to handle exceptions. This mode only runs in Privileged level.
- Privileged levels
  - User level: The software has limited access.
  - Privileged level: The software can use all instructions and access all resources.

Privileged level is switched by the CPU-specific CONTROL register. Switching from Privileged level to User level is performed by the CONTROL register. However, the CONTROL register can be rewritten only with the privileged level. Therefore, switching from the User level to the Privileged level must always go through the Handler mode. The CPU enters the Handler mode when an exception or interrupt occurs. Figure 5 shows an example User/Privileged level switching by the supervisor call (SVC) instruction exception. The SVC instruction generates an exception and can enter the Handler mode.

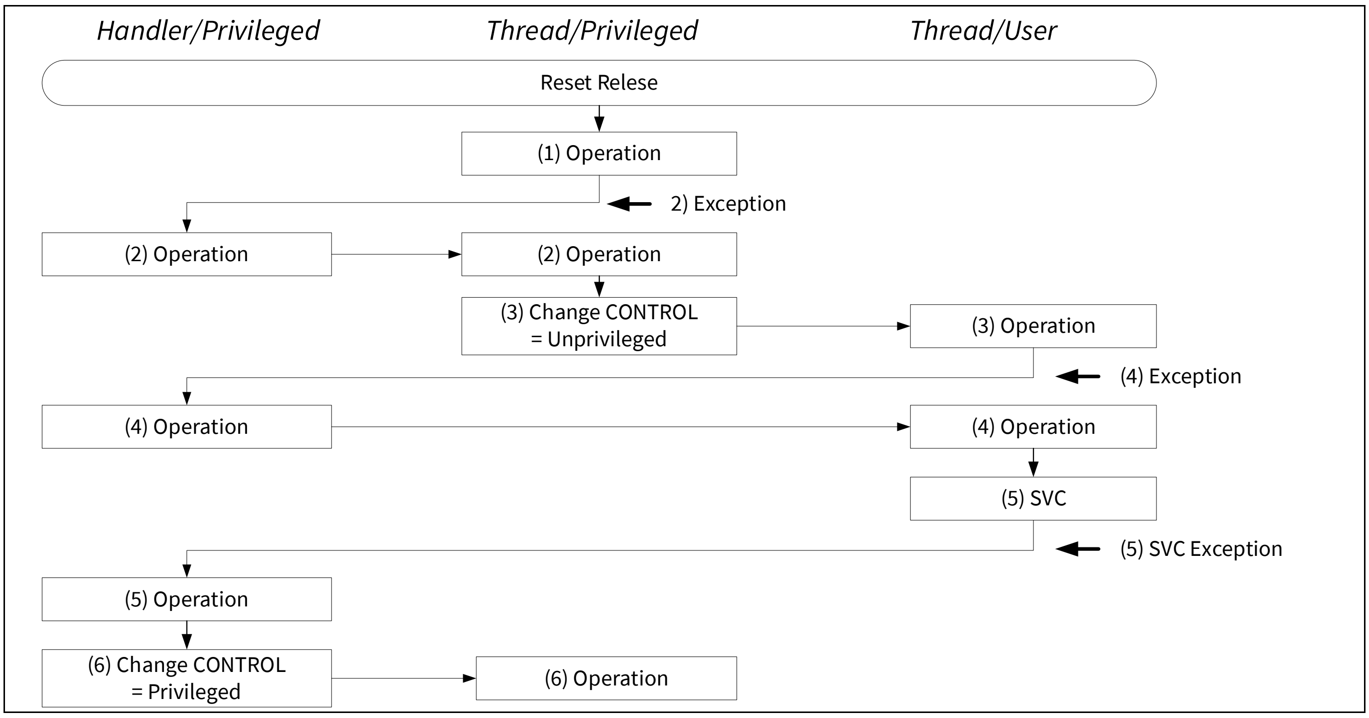

Figure 5 Example User/Privileged level switching for both CPUs

1. CPUs are started in Thread/Privileged mode after reset release.
2. When an exception occurs in Thread/Privileged mode, the Handler/Privileged mode is entered; upon returning from handler processing, Thread/Privileged mode is entered again.
3. In the Thread/Privileged mode, transition to the Thread/User mode is allowed by the CONTROL register.
4. When an exception occurs in the Thread/User mode, the Handler/Privileged mode is entered; upon returning from handler processing, Thread/User mode is entered again.
5. When switching from the Thread/User mode to Thread/Privileged mode, use the SVC instruction to enter the Handler/Privileged mode. The SVC instruction can cause an SVC exception.
6. Set the Privileged level with the CONTROL register in the Handler/Privileged level. The CPU transitions to the Thread/Privileged mode after returning from handler processing.

See the Arm® documentation sets for CM7 and CM0+ for more details.

You need to register the SVC handler in advance.

3.4 Protection context attribute setting

Protection contexts (PCs) are used to isolate software execution for security and safety purposes. PCs are used as the PC attribute for all bus transfers that are initiated by the master. SMPUs and PPUs allow or restrict bus transfers based on the PC attribute.

The series supports eight PCs. Protection contexts 0 and 1 out of eight PCs are special; these are controlled by hardware. In addition, PC0 has unrestricted access.

Specific bus masters have associated PC fields (PROT_MPUx_MS_CTL.PC and PROT_SMPU_MSx_CTL.PC_MASK_15_TO_1 and PC_MASK_0).

A bus master protection context is changed by reprogramming the associated PROT_MPUx_MS_CTL.PC field. The PROT_SMPU_MSx_CTL.PC_MASK field restricts the PCs that can be set for the associated bus master.

For example, if PROT_SMPU_MSx_CTL.PC_MASK[15:0] = “0x06” (PC1, 2 = "1"), the PCs to which the associated bus master can be set are “PC = 1” and “PC = 2”. A bus master cannot be changed to a PC not allowed (PC = 0, 3, 4, 5, 6, 7).

Figure 6 shows an example of changing the flow of PCs.

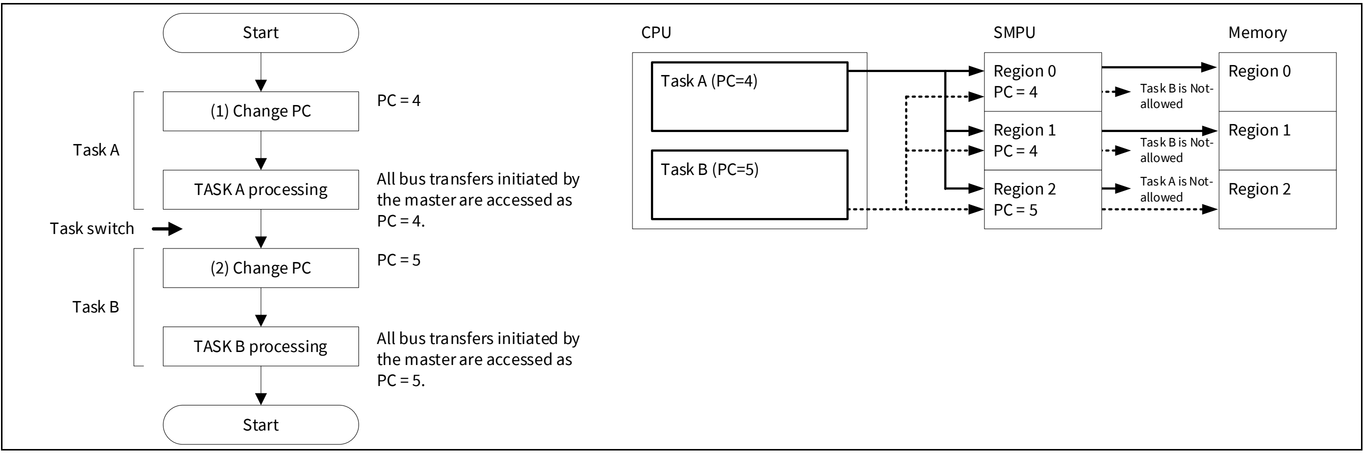

Figure 6 Change flow of PCs and behavior

Note: *PC values that can be set by each master are restricted by PROT_SMPU_MSx_CTL.PC_MASK_15_TO_1 and PC_MASK_0.*

Note: *If the access attributes such as PC or secure change when the CM7 cache memory is enabled, data that is not allowed access such as the new PC or non-secured may be stored in the cache memory. In this case, you need to run the cache memory clean and invalidate before switching the attribute of the protection context and secured. See AN234254-Multi-core handling guide in XMC7000 for cache memory handling.*

This allows a single bus master to take on different protection roles by reprogramming only the protection context field without changing the settings of SMPUs and PPUs.

3.5 Bus transfer evaluation

3.5.1 Evaluation process

The evaluation of bus transfer by protection units is divided into two independent processes:

- Matching process: For each protection structure, this process determines whether a transfer address is contained within the address range.
- Access evaluation process: For each protection structure, this process evaluates the bus transfer access attributes against the access control attributes.

The following pseudo code shows the evaluation process of bus transfer.

```
match = 0;
for (i = n-1; i >= 0; i--)       // n: number of protection regions
   if (Match ("transfer address", "protection context") {
      match = 1; break;
   }
                                                          Matching Process

if (match)
   AccessEvaluate ("access attributes", "protection context");
else
   "access allowed"
                                                          Access Evaluation Process
```

Note: *If no protection structure provides a match, access is allowed.*

Note: *If multiple protection structures provide a match, the access control attributes for access evaluation are provided by the protection structure with the highest index.*

A protection unit evaluates the protection structures in the decreasing order. In other words, higher-indexed structures take precedence over lower-indexed structures.

When transfer addresses do not match, the protection structure with the next highest index is evaluated. When transfer addresses match, bus transfer access attributes are evaluated by the access evaluation process. If transfer access attributes do not match with the access evaluation process, it is detected as an access violation. Therefore, the protection structure with the next highest index is not evaluated.

3.5.2 PC_MATCH operation

The SMPU has a PC_MATCH field, which controls "matching" and "access evaluation" processes.

- Case of PC_MATCH = 0

The following pseudo code shows the evaluation process when PC_MATCH = 0:

```
match = 0;
for (i = n-1; i >= 0; i--)       // n: number of protection regions
   if (Match (“transfer address”) {
      match = 1; break;
   }

if (match)
   AccessEvaluate (“access attributes”, “protection context”);
else
   “access allowed”
```

When PC_MATCH = “0”, the protection context is evaluated only in the access evaluation process.

Figure 7 shows the operation example when PC_MATCH of Region 5 is ‘0’ and PC_MATCH of Region 4 is ‘0’. Table 2 shows the settings for each region.

Table 2 Region Setting1 for PC_MATCH operation

Region	PC_MATCH	Region address	Protection context	User
Region 4	0	AA	4	Read/Write
Region 5	0	AA	5	Read-only

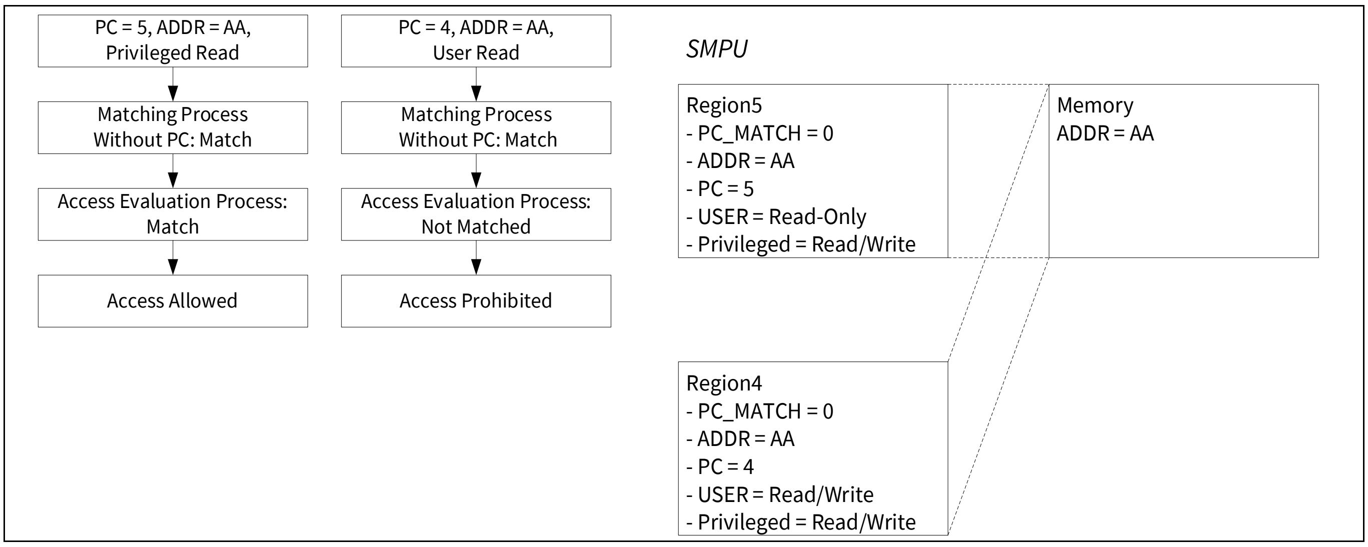
PC = 5, ADDR = AA, Privileged Read
Matching Process Without PC: Match
Access Evaluation Process: Match
Access Allowed
PC = 4, ADDR = AA, User Read
Matching Process Without PC: Match
Access Evaluation Process: Not Matched
Access Prohibited
SMPU
Region5 - PC_MATCH = 0 - ADDR = AA - PC = 5 - USER = Read-Only - Privileged = Read/Write
Memory ADDR = AA
Region4 - PC_MATCH = 0 - ADDR = AA - PC = 4 - USER = Read/Write - Privileged = Read/Write

Figure 7 PC_MATCH operation example 1

In this case, the protection context is not evaluated by the matching process. Therefore, both PC = 4 access and PC = 5 access is "match" in the matching process. The protection context is evaluated by the access evaluation process. For PC = 5, access is allowed; for PC = 4, access is not allowed. As a result, PC = 4 access cannot access this address because PC = 4 access is prohibited by Region 5 with a higher priority than Region 4.

- Case of PC_MATCH = 1

The following pseudo code shows the evaluation process when PC_MATCH = 1:

```
match = 0;
for (i = n-1; i >= 0; i--)        // n: number of protection regions
   if (Match (“transfer address”, “protection context”) {
      match = 1; break;
   }

if (match)
   AccessEvaluate (“access attributes”, “protection context”);
else
   “access allowed”
```

When PC_MATCH = "1", the protection context is evaluated not only by the access evaluation process, but also by the matching process.

Figure 8 shows the operation example when PC_MATCH of Region5 is ’1’ and PC_MATCH of Region4 is ‘0’. Table 3 shows the settings for each region.

Table 3 Region Setting2 for PC_MATCH operation

Region	PC_MATCH	Region address	Protection context	User
Region 4	0	AA	4	Read/Write
Region 5	1	AA	5	Read only

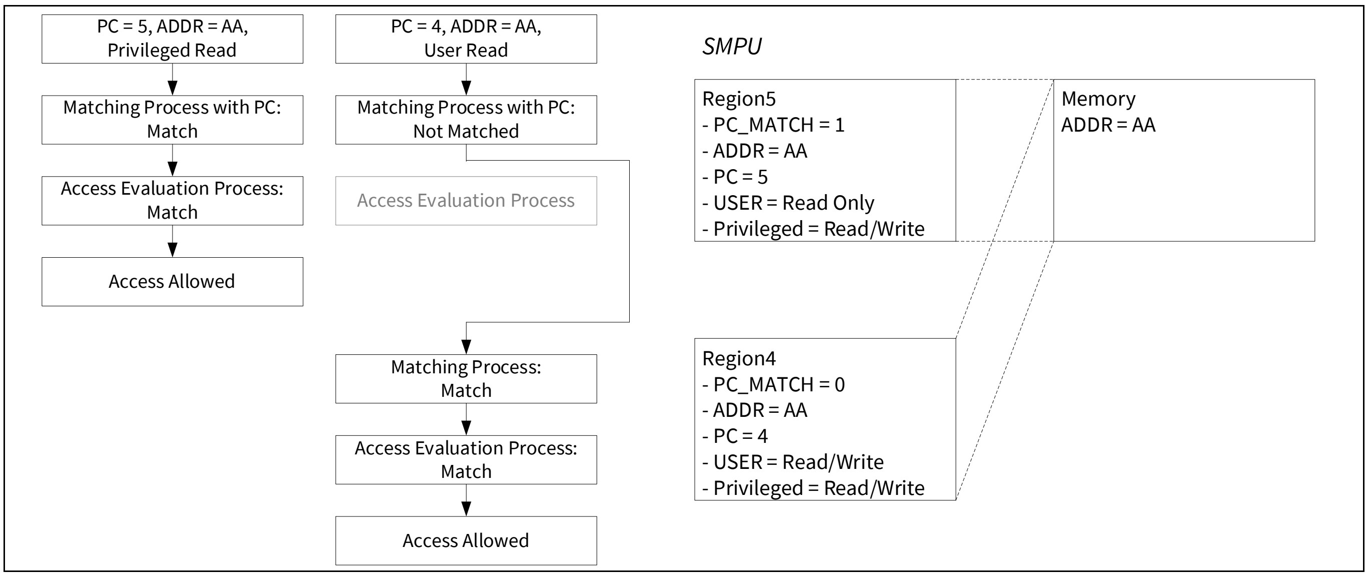
PC = 5, ADDR = AA, Privileged Read
Matching Process with PC: Match
Access Evaluation Process: Match
Access Allowed
PC = 4, ADDR = AA, User Read
Matching Process with PC: Not Matched
Access Evaluation Process
Matching Process: Match
Access Evaluation Process: Match
Access Allowed
SMPU
Region5 - PC_MATCH = 1 - ADDR = AA - PC = 5 - USER = Read Only - Privileged = Read/Write
Region4 - PC_MATCH = 0 - ADDR = AA - PC = 4 - USER = Read/Write - Privileged = Read/Write
Memory ADDR = AA

Figure 8 PC_MATCH operation example 2

In this case, the protection context is also evaluated by the matching process. PC = 5 access is "match" in the matching process and is evaluated by the access evaluation process. However, PC = 4 access is "Not matched", and is evaluated by the matching process with Region 4 of the next highest priority. It is evaluated by the access evaluation process after the matching process, and access is allowed.

It is possible to assign different attributes to the same address area depending on the protection context by using PC_MATCH.

Note: *PC_MATCH is provided only for the SMPU. This functionality is not supported because a PPU structure provides access attributes for all protection contexts.*

Note: *When the region address has only one attribute, use PC_MATCH with "0".*

3.6 Master identifier

Each bus master has a dedicated master identifier. This identifier is used for correspondence with the register suffix in protection units and identification of access violation master by protection units. Table 4 lists the master identifiers. See the related datasheet [1] for bus master identifier using products.

Table 4 **Master identifiers**

Master identifier	**Bus master**	**Bus master**
	XMC7100 series	**XMC7200 series**
0	CM0+ CPU	CM0+ CPU
1	CRYPTO component	CRYPTO component
2	P-DMA (DW) 0	P-DMA (DW) 0
3	P-DMA (DW) 1	P-DMA (DW) 1
4	M-DMA (DMAC)	M-DMA (DMAC)
5	SDHC	SDHC
6	Ethernet 0	–
9	–	Ethernet 0
10	–	Ethernet 1
13	CM7_1 CPU	CM7_1 CPU
14	CM7_0 CPU	CM7_0 CPU
15	DAP Tap Controller	DAP Tap Controller

3.7 Protection violation

If the MPU that is implemented as part of the CPU detects access violations, the programmable-priority MemManage fault or HardFault handler is invoked. If an MPU fault occurs on an access that is not in the TCM, the AXI or AHB transactions for that access are not performed. See the Arm® documentation sets for CM7, and CM0+ for more MPU details.

If the MPU that is implemented as part of the bus infrastructure and SMPU detects a bus transfer that causes a violation of the protection status, the bus transfer results in a bus error.

In case of write transfers that violate PPU protection, the bus master will not see the bus error when buffering is enabled (CPUSS_BUFF_CTL.WRITE_BUFF = 1). This is because AHB-Lite bridges in the bus infrastructure will buffer the write transfer and send the OK response to masters. In this case, the system must depend on the fault reported by the PPU. Write transfers that violate the PPU cause a bus error if buffering is disabled (CPUSS_BUFF_CTL.WRITE_BUFF = 0). Read transfers that violate PPU protection always result in a bus error.

The bus transfers that violate protection units do not reach their target memory location or peripheral register.

A protection violation detected by the MPU that is implemented as part of the bus infrastructure, SMPU, and PPU is captured in the fault report structure. The fault report structure can generate an interrupt to indicate the occurrence of a fault. In addition, information on the violating bus transfer is communicated to the fault report structure.

The fault reporting structure captures the following information:

- Violating address
- Violating attribute
  - User Read/User Write/User Execute
  - Privileged Read/Privileged Write/Privileged Execute
  - Non-secured
  - Protection context identifier
- Violating master identifier
- Protection units that detected the violation or fault type[1]

[1] Depends on the detected fault. See the registers reference manual.

4 Protection unit structure

4.1 MPU structure

Figure 9 shows the MPU structure that is implemented as part of the bus infrastructure. See the Arm® documentation sets for CM7 and CM0+ for details of the MPU that is implemented as part of CM7 and CM0+.

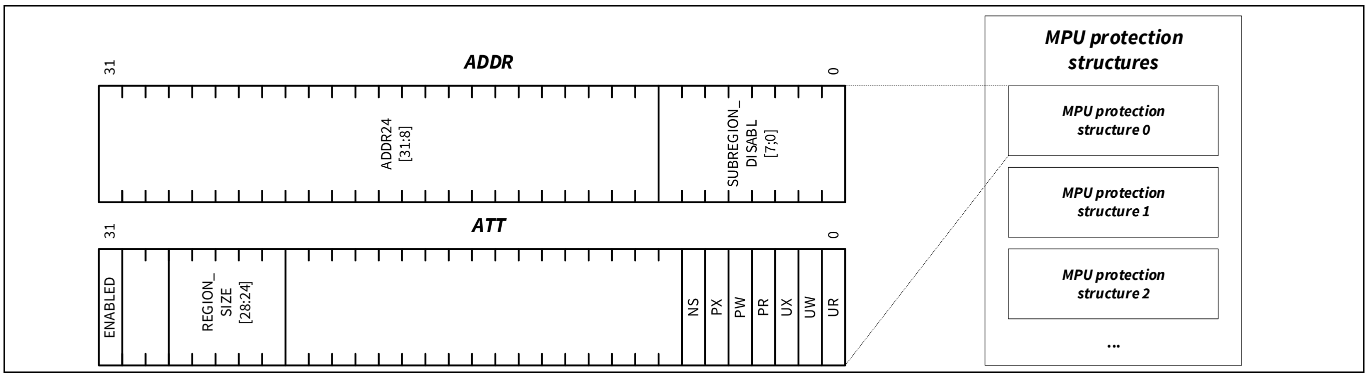
ADDR
31
0
ADDR24 [31:8]
SUBREGION_DISABL [7;0]
ATT
31
0
ENABLED
REGION_SIZE [28:24]
NS
PX
PW
PR
UX
UW
UR
MPU protection structures
MPU protection structure 0
MPU protection structure 1
MPU protection structure 2
...

Figure 9 MPU structure

The MPU protection structure sets the property to be allowed and restricted by master access. An MPU protection identifies the following properties:

- Address range
  - ADDR.ADDR24 [31:8]: Specifies the base address of the region
  - ATT.REGION_SIZE [28:24]: Specifies the size of a region. The region size is in the range of [256 B, 4 GB].
  - ADDR.SUBREGION_DISABLE [7:0]: Individual disable settings for eight subregions within the region
- Access attribute
  - ATT.UR: Control for User Read access
  - ATT.UW: Control for User Write access
  - ATT.UX: Control for User Execute access
  - ATT.PR: Control for Privileged Read access
  - ATT.PW: Control for Privileged Write access
  - ATT.PX: Control for Privileged Execute access
  - ATT.NS: Control for Secured access
- Region enable
  - ATT.ENABLED: Region enable

The MPU does not provide a protection context. The definition of this MPU type follows the Arm® MPU definition (in terms of memory region and access attribute definition) to ensure a consistent software interface.

A region can be partitioned into eight equally sized subregions; it is possible to specify individual enables for the subregions within a region with the ADDR.SUBREGION_DISABLE field.

For example, when SUBREGION_DISABLE is 0x82 (bit fields 1 and 7 are '1') in the divided subregion [0: 7], subregion1, 7 are disabled, subregions 0, 2, 3, 4, 5, and 6 are enabled. Table 5 shows the areas of the eight subregions and the Enable/Disable states if the start address is 0x10005400, and the region ranges from 0x10005400 to 0x100055ff (512 bytes).

Table 5 Each subregion area and state

Subregion	Area	State
Subregion 0	0x10005400 to 0x1000543f	Enable
Subregion 1	0x10005440 to 0x1000547f	Disable
Subregion 2	0x10005480 to 0x100054bf	Enable
Subregion 3	0x100054c0 to 0x100054ff	Enable
Subregion 4	0x10005500 to 0x1000553f	Enable
Subregion 5	0x10005540 to 0x1000557f	Enable
Subregion 6	0x10005580 to 0x100055bf	Enable
Subregion 7	0x100055c0 to 0x100055ff	Disable

4.2 SMPU structure

Figure 10 shows the SMPU structure.

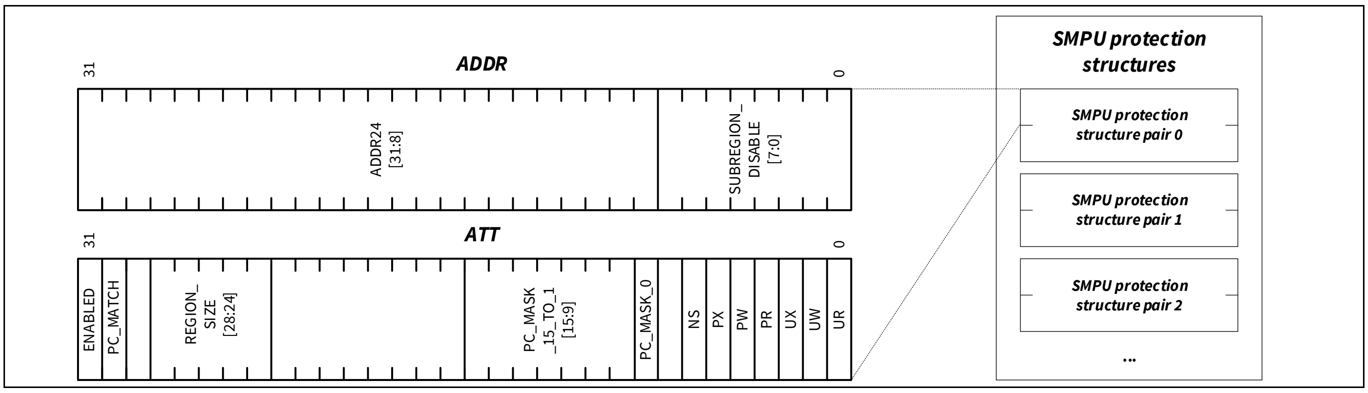

Figure 10 SMPU structure

The SMPU protection structure sets the property to be allowed and restricted by master access. SMPU protection identifies the following properties:

- Address range
  - ADDR.ADDR24 [31:8]: Specifies the base address of a region
  - ATT.REGION_SIZE [28:24]: Specifies the size of a region. The region size is in the range of [256 B, 4 GB].
  - ADDR.SUBREGION_DISABLE [7:0]: Individual disables for eight subregions within the region
- Access attribute
  - ATT.UR: Control for User Read access
  - ATT.UW: Control for User Write access
  - ATT.UX: Control for User Execute access
  - ATT.PR: Control for Privileged Read access
  - ATT.PW: Control for Privileged Write access
  - ATT.PX: Control for Privileged Execute access
  - ATT.NS: Control for Secured access
  - ATT.PC_MASK_15_TO_1 and PC_MASK_0: Control for individual protection contexts
  - The PC_MASK_0 field is always ‘1’. In other words, PC=0 is always allowed.

- ATT.PC_MATCH: Specifies whether the PC field participates in the "matching" process or the "access evaluation" process. See PC_MATCH operation for more details.
- Region enable
  - ATT.ENABLED: Region enable

The SUBREGION function of SMPU is the same as that of the MPU.

4.3 PPU structure

Figure 11 shows the PPU structure.

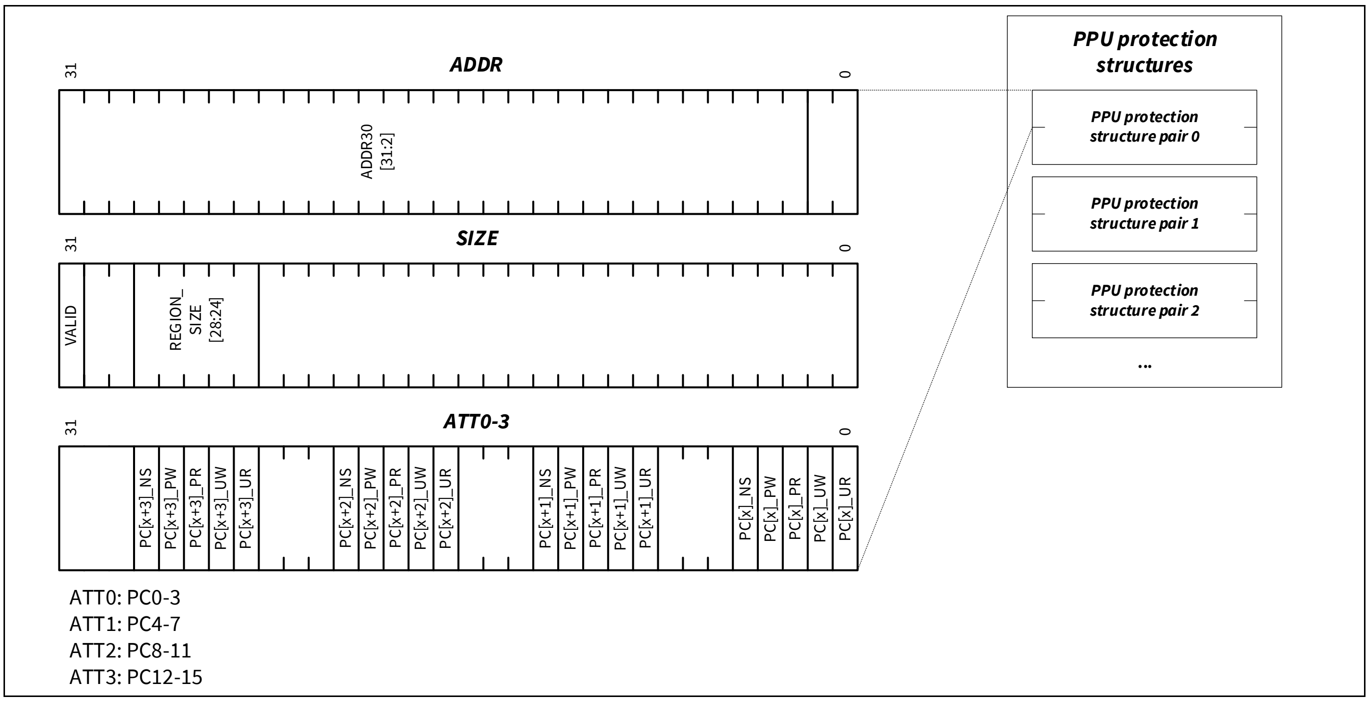

Figure 11 PPU structure

The PPU protection structure sets the property to be allowed and restricted by master access. The PPU can have independent attribute settings for all protection context attributes. PPU protection identifies the following properties:

- Address range
  - ADDR.ADDR30 [31:2]: Specifies the base address of a region
  - SIZE.REGION_SIZE [28:24]: Specifies the size of a region. The region size is in the range of [4 B, 2 GB].

In the Fixed PPU structure, it has a fixed, constant address region.

- Access attribute
  - ATT.PCx_UR: Control for PCx User Read access
  - ATT.PCx_UW: Control for PCx User Write access
  - ATT.PCx_PR: Control for PCx Privileged Read access
  - ATT.PCx_PW: Control for PCx Privileged Write access
  - ATT.PCx_NS: Control for PCx Secure access
- Region enable

- SIZE.VALID: Region enable

Note: In the series, the protection context is supported from 0 to 7. Therefore, only ATT0,1 are present.

4.4 Protection pair structure

Registers of protection units are the same registers as other peripherals. Furthermore, the registers of a protection structure can be included in the address range of another protection structure as with peripherals. Therefore, a protection structure can be protected by another protection structure.

The protection structure that protects a protection structure is called the master structure, and the protection structure to be protected by master is called the slave structure. The Slave structure protects peripherals.

The protection structure of a slave and master is referred to as a protection pair. SMPUs and PPUs have protection pairs. Figure 12 shows the protection pair structure.

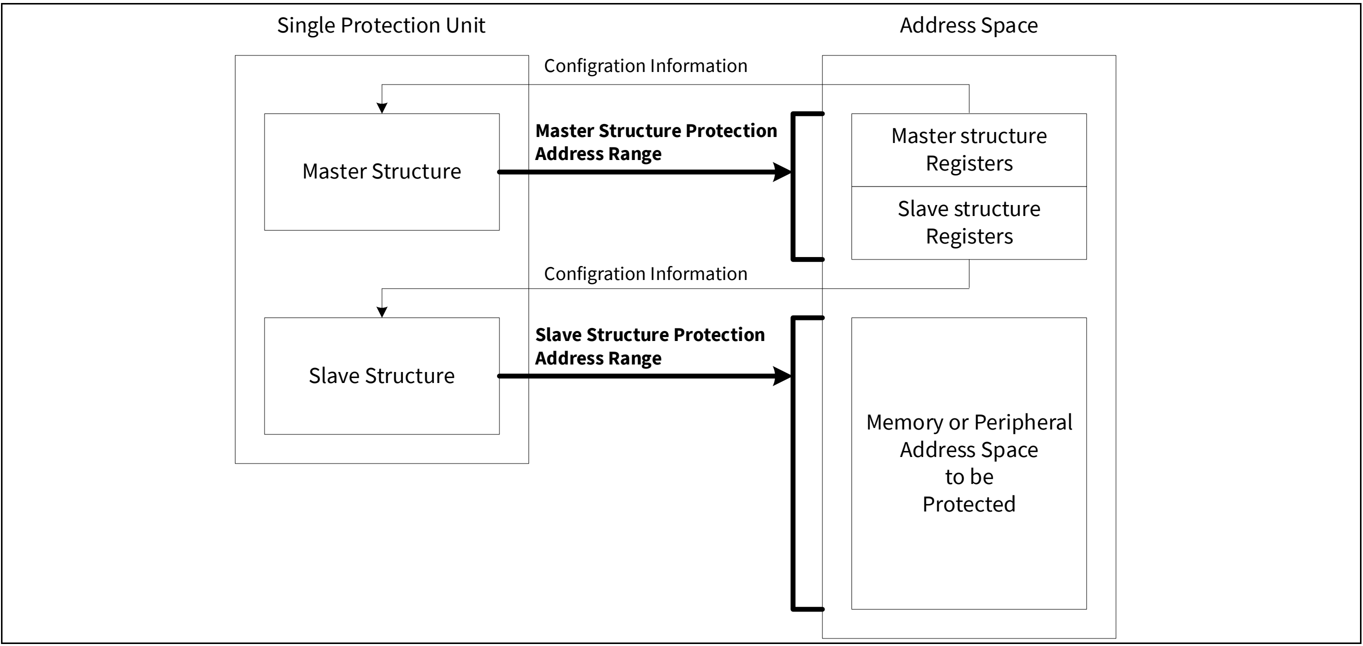
Single Protection Unit
Address Space
Configration Information
Master Structure
Master Structure Protection Address Range
Master structure Registers
Slave structure Registers
Configration Information
Slave Structure
Slave Structure Protection Address Range
Memory or Peripheral Address Space to be Protected

Figure 12 Protection pair structure

The master structure that protects the slave protection structure has the following features:

- Address range
  - ADDR.ADDR: Read-only; it has a fixed, constant address region.
  - ATT.REGION_SIZE: Read-only; it has a fixed, constant address region.
  - ADDR.SUBREGION_DISABLE: Read-only; it has a fixed, constant address region.
- Access attribute
  - ATT.UR: Fixed to ‘1’; User Read accesses are always allowed
  - ATT.UW: Control for User Write access
  - ATT.UX: Fixed to ‘0’; User execute accesses are never allowed
  - ATT.PR: Fixed to ‘1’; Privileged Read accesses are always allowed
  - ATT.PW: Control for Privileged Write access
  - ATT.PX: Fixed to ‘0’; Privileged Execute accesses are never allowed

- ATT.NS: Control for Secured access

The above is an example of the SMPU master structure.

In the master structure, the protected region is fixed; read access is always allowed, and execution access is not allowed. SMPUs can be enabled or disabled, but PPUs cannot be disabled.

5 Configuration example of protection units

An example of using the protection units is explained according to the following usage assumptions.

Note: *The addresses and peripheral channel numbers shown in this section are those of the XMC7200 series. See the technical reference manual [2] for the actual addresses and peripheral channel numbers.*

5.1 Configuration example: MPU implemented as part of the CPU

This section explains how to protect the area used by the operating system (OS) from tasks accesses, and shows an example of the configuration of the MPU.

5.1.1 Use case

This section provides configuration examples for the MPU. An MPU distinguishes between User/Privileged, Read/Write, and Execute accesses. Table 6 shows the access restriction for the MPU.

Table 6 Example access restriction for MPU implemented as part of the CPU

Region	Attribute
Region 0 (Background address) Base address: 0x00000000 Size: 4 GB	Privileged: Read/Write User: Read/Write Execution is permitted
Region 1 (Code flash) Base address: 0x10000000 Size: 8 MB	Privileged: Read only User: Read only Execution is permitted
Region 2 (Work flash) Base address: 0x14000000 Size: 256 KB	Privileged: Read only User: No access Execution is not permitted
Region 3 (SRAM) Base address: 0x28000000 Size: 1 MB	Privileged: Read/Write User: Read/Write Execution is permitted
Region 4 (Peripheral registers) Base address: 0x40000000 Size: 64 MB	Privileged: Read/Write User: Read/Write Execution is not permitted
Region 5 (System registers for ARM) Base address: 0xE0000000 Size: 512 MB	Privileged: Read/Write User: Read/Write Execution is not permitted
Other regions (Unused)	-

Region 0 is used as the background region. If no attributes are specified by another region, the background region will be applied.

Note that when the MPU Implemented as part of the CPU is enabled, access to unconfigured areas will cause access violations. To prevent unintended access violations, this type of MPU supports overlapping. Therefore, the highest region number has the highest priority, while the lowest region number (Region 0) has the lowest priority. Attributes of a region that overlaps Region 0 have a higher priority than attributes of Region 0. That is, Region 0 can be used as a background region (determination of the attributes of all areas).

See the Arm® documentation sets for CM7, and CM0+ for more MPU details.

Region 1 is read-only for both Privileged and User access. Execution is allowed, but data cannot be written. Region 2 is read-only for Privileged only. You cannot access it. Data cannot be written by both Privileged and User access, and execution is not allowed. Region 3 can be accessed from either Privileged or User, and execution is allowed. Region 4 and 5 can be accessed from either Privileged or User, and execution is not allowed.

5.1.2 Setting procedure

This type of MPU is set by CPU-specific registers. The MPU must be disabled when it is set, and it is necessary to set the MPU in the Privileged level. Figure 13 shows an example of how to set an MPU.

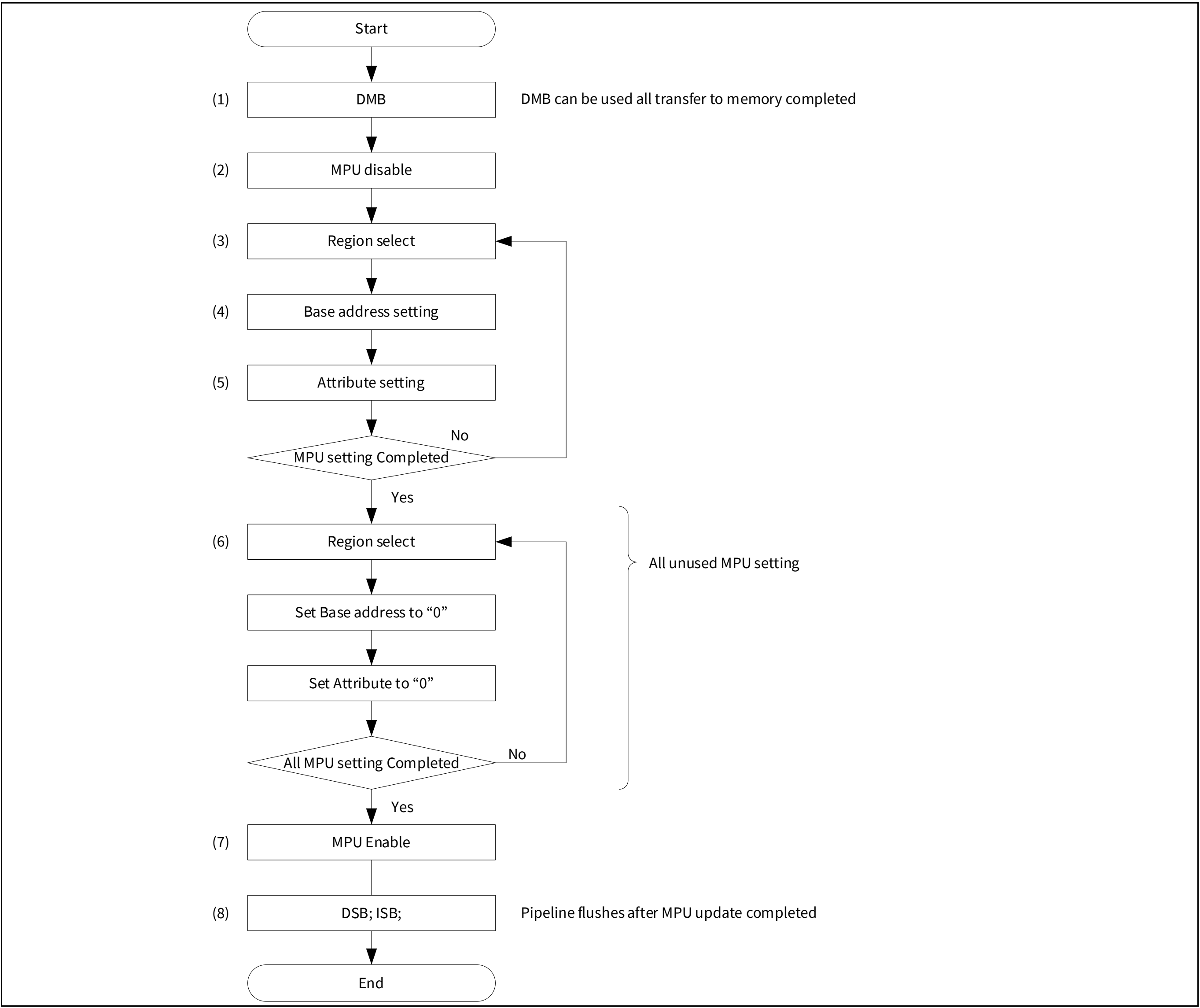

Figure 13 Procedure example for setting MPU implemented as part of the CPU

The MPU must be disabled when setting it. First, specify the region to be set with the MPU_RNR register. After setting MPU_RNR, set the base address, region size, and access attribute with the MPU_RBAR and MPU_RASR registers. Repeat this for each region. Also, MPU_RBAR and MPU_RASR registers of unused regions are set to '0'. Finally, the MPU is enabled.

It is also possible to specify the region number to set the MPU directly with the MPU_RBAR register. MPU_RASR also be used to set the subregion and memory attributes such as Normal memory, Strongly-ordered, and Device.

See the Arm® documentation sets for CM7, and CM0+ for more details.

5.2 Configuration of MPU implemented as part of bus infrastructure

This type of MPU is set by the MPU.ADDR and MPU.ATT registers and is used by a test controller. However, normally, this MPU is set with CM0+ as a secure CPU depending on security requirements.

This MPU is set in the boot process.

5.3 Configuration example of SMPU

The SMPU provides the memory protection function and is shared by all bus masters. All bus masters have the same restriction for each region.

The SMPU has a protection pair structure with master/slave. Therefore, the setting the Slave structure attribute is restricted by the master structure setting.

5.3.1 Usage assumptions

The SMPU distinguishes User/Privileged, Secure/Non-secure, and protection contexts.

Table 7 shows an example of access restriction for an SMPU.

Table 7 Example access restriction for SMPU

Region	Privileged	User	Secure	Allowed protection context	PC_MATCH	Resources
Region 2 Base address: 0x28019000 Size: 4 KB	Read/Write Execution is permitted	Read/Write Execution is not permitted	Non-secure	PC = 6	Access evaluation	SRAM
Region 3 Base address: 0x28018000 Size: 4 KB	Read/Write Execution is permitted	Read/Write Execution is not permitted	Non-secure	PC = 5	Access evaluation	SRAM

Regions 2 and 3 have access restricted by the protection context.

Region 2 can access with protection context = 6, and Region 3 can access with protection context = 5.

5.3.2 Setting procedure for SMPU

Figure 14 shows an example of the setting procedure.

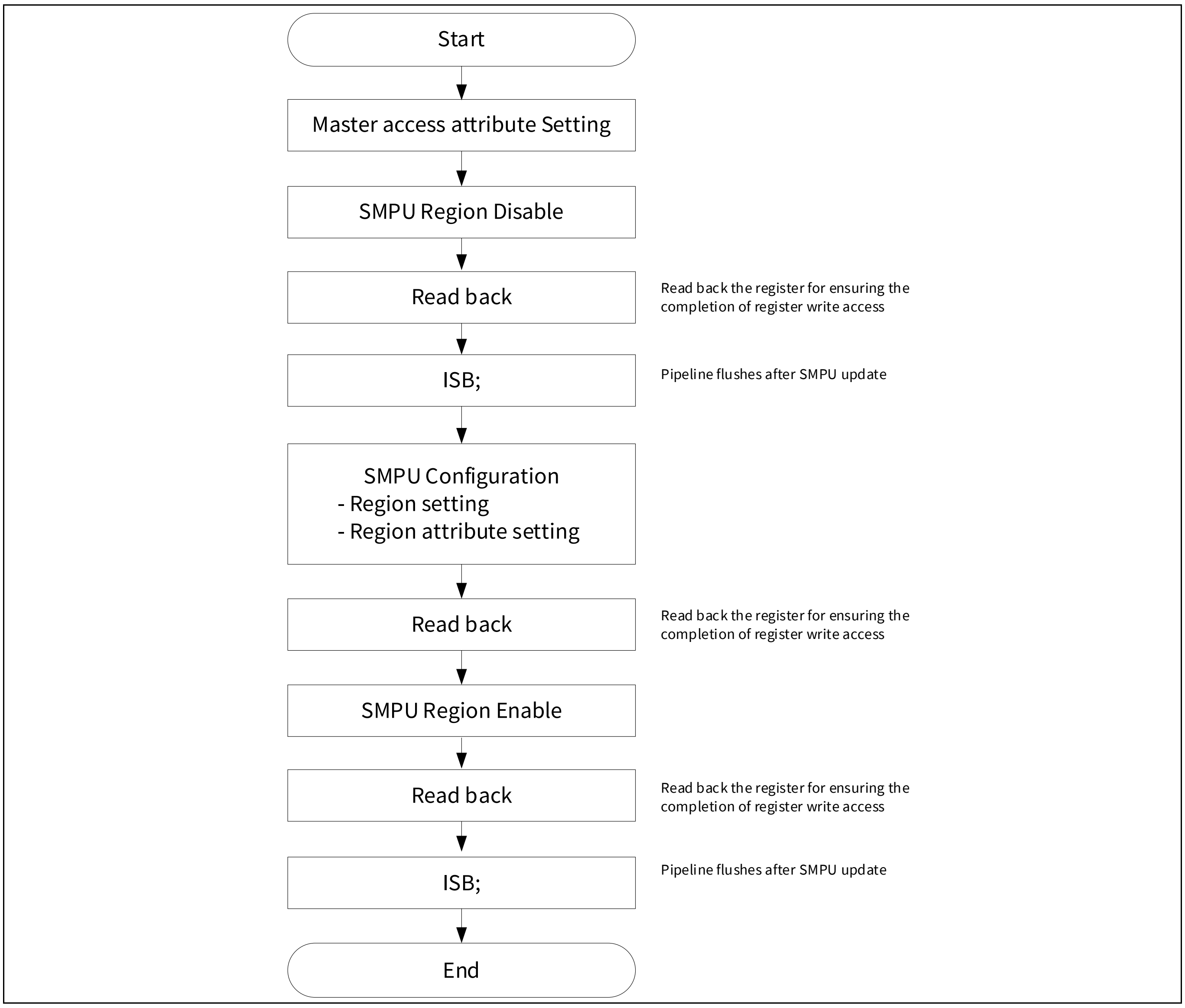
Start
Master access attribute Setting
SMPU Region Disable
Read back
Read back the register for ensuring the completion of register write access
ISB;
Pipeline flushes after SMPU update
SMPU Configuration
- Region setting
- Region attribute setting
Read back
Read back the register for ensuring the completion of register write access
SMPU Region Enable
Read back
Read back the register for ensuring the completion of register write access
ISB;
Pipeline flushes after SMPU update
End

Figure 14 Procedure example for setting SMPU

The access attribute for setting the slave structure (PROT_SMPU_STRUCTx_ADDR0 and PROT_SMPU_STRUCTx_ATT0) is allowed by the master structure (PROT_SMPU_STRUCTx_ADDR1 and PROT_SMPU_STRUCTx_ATT1).

It is necessary to read back the register for ensuring the completion of register write access when SMPU setting is completed.

5.3.3 Configuration

Table 8 and Table 9 list the parameters and functions of the configuration part in PDL for SMPU configuration.

Table 8 Parameters

Parameters	Description	Value
MASTER_ID_OF_THIS_CPU	Defines the master for which PROT_SMPU_MSx_CTL is set	CPUSS_MS_ID_CM0 (CM0+)
TP_PRIVILEGED	Defines the PROT_SMPU_MSx_CTL.P value	1ul (Privileged mode)
TP_SECURE	Defines the PROT_SMPU_MSx_CTL.NS value	0ul (Non-Secure)
TP_PROT_CONTEXT	Defines the PROT_SMPU_MSx_CTL.PC value	6ul
TP_PERMITTED_ADDR	Defines the base address of SMPU Region 2	0x28019000UL
TP_PERMITTED_CONTEXT	Defines the PC value that can access Region 2	6ul
TP_PROHIBITED_ADDR	Defines the base address of SMPU Region 3	0x28018000UL
TP_PROHIBITED_CONTEXT	Defines the PC value that can access Region 3	5ul
smpuStruct(2 or 3)Config.address	Sets Region 2 or 3 base address	Region 2: TP_PERMITTED_ADD Region 3: TP_PROHIBITED_ADDR
smpuStruct(2 or 3)Config.regionSize	Sets Region 2 or 3 size	CY_PROT_SIZE_4KB (4 KB)
smpuStruct(2 or 3)Config.subregions	Sets Region 2 or 3 sub-region setting	0x00ul (Unused)
smpuStruct(2 or 3)Config.userPermission	Sets Region 2 or 3 access attribute for user	CY_PROT_PERM_RW (Execution is not permitted)
smpuStruct(2 or 3)Config.privPermission	Sets Region 2 or 3 access attribute for privileged	CY_PROT_PERM_RWX (Full access)
smpuStruct(2 or 3)Config.secure	Sets Region 2 or 3 access attribute for non-secure	0ul (Non-secure)
smpuStruct(2 or 3)Config.pcMatc	Sets Region 2 or 3 PC_MATCH setting	0ul (Access evaluation)
smpuStruct(2 or 3)Config.pcMask	Sets Region 2 or 3 PC_MASK setting	Region 2: 1ul << (TP_PERMITTED_CONTEXT - 1) (Protection context = 6 is permitted) Region 3: 1ul << (TP_PROHIBITED_CONTEXT - 1)

Parameters	Description	Value
		(Protection context = 5 is permitted)
PROT_SMPU_SMPU_STRUCT2	Defines the base address SMPU structure registers (region 2)	0x40232080ul
PROT_SMPU_SMPU_STRUCT3	Defines the base address SMPU structure registers (region 3)	0x402320C0ul

Table 9 Functions

Functions	Description	Value
`Cy_Prot_ConfigBusMaster(busMaster, privileged, secure, pcMask)`	Configures the PROT_SMPU_MS0_CTL register busMaster; indicate setting register number privileged: P field setting value secure: NS field setting value pcMask: Specifies a protection context value that can be set by the associated master.	busMaster; MASTER_ID_OF_THIS_CPU privileged: TP_PRIVILEGED secure: TP_SECURE pcMask: 1 << (TP_PROT_CONTEXT-1) (Protection context = 6)
`Cy_Prot_DisableSmpuSlaveStruct(*base)`	SMPU region disable *base: Base address of the SMPU structure	*base: PROT_SMPU_SMPU_STRUCT2 or PROT_SMPU_SMPU_STRUCT3
`Cy_Prot_EnableSmpuSlaveStruct(*base)`	SMPU region enable *base: Base address of the SMPU structure	*base: PROT_SMPU_SMPU_STRUCT2 or PROT_SMPU_SMPU_STRUCT3
`Cy_Prot_ConfigSmpuSlaveStruct(*base, config)`	Configures the SMPU structure *base: Base address of the SMPU structure Config: Configuration data	*base: PROT_SMPU_SMPU_STRUCT2 or PROT_SMPU_SMPU_STRUCT3 Config: smpuStruct(2 or 3)Config

Code Listing 1 shows an example of SMPU configuration.

Code Listing 1 Example of SMPU configuration

```
/*************************************************************************
*****
* Macros
*************************************************************************
****/
#if (CY_CPU_CORTEX_M0P)
    #define MASTER_ID_OF_THIS_CPU CPUSS_MS_ID_CM0
#else
    #define MASTER_ID_OF_THIS_CPU CPUSS_MS_ID_CM7_0
#endif /* (CY_CPU_CORTEX_M0P) */
```

Code Listing 1 **Example of SMPU configuration**

```
#define TP_PRIVIREGED          (1)              /* privileged */
#define TP_SECURE              (0)              /* non secure */
#define TP_PROT_CONTEXT        (6)              /* enable context 6 */

/* This area is going to be prohibited accessing from a master who has
TP_PROHIBITED_CONTEXT as context */
#define TP_PROHIBITED_ADDR     (0x28018000UL)
#define TP_PROHIBITED_CONTEXT  (5)

/* This area is going to be permitted accessing from a master who has
TP_PERMITTED_CONTEXT as context */
#define TP_PERMITTED_ADDR      (0x28019000UL)
#define TP_PERMITTED_CONTEXT   (TP_PROT_CONTEXT)

const cy_stc_smpu_cfg_t smpuStruct2Config =
{
    .address        = (uint32_t*)(TP_PERMITTED_ADDR),
    .regionSize     = CY_PROT_SIZE_4KB,                     // 4KB:
0x1000 Byte
    .subregions     = 0x00,
    .userPermission = CY_PROT_PERM_RWX,
    .privPermission = CY_PROT_PERM_RWX,
    .secure         = 0,                                    // Non secure
    .pcMatch        = 0,
    .pcMask         = 1 << (TP_PERMITTED_CONTEXT - 1),      // enable
context "TP_PERMITTED_CONTEXT"
};

const cy_stc_smpu_cfg_t smpuStruct3Config =
{
    .address        = (uint32_t*)(TP_PROHIBITED_ADDR),
    .regionSize     = CY_PROT_SIZE_4KB,                      // 4KB:
0x1000 Byte
    .subregions     = 0x00,
    .userPermission = CY_PROT_PERM_RWX,
    .privPermission = CY_PROT_PERM_RWX,
    .secure         = 0,                                     // Non
secure
```

Code Listing 1 **Example of SMPU configuration**

```
    .pcMatch        = 0,
    .pcMask         = 1 << (TP_PROHIBITED_CONTEXT - 1),        // enable
context "TP_PERMITTED_CONTEXT"
};

int main(void)
{
    /* Variable to capture return value of functions */
      cy_rslt_t result = CY_RSLT_SUCCESS;

      cy_en_prot_status_t status;

    /* Initialize the device and board peripherals */
    result = cybsp_init();

    /* Board init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Enable global interrupts */
    __enable_irq();

    /* Initialize retarget-io to use the debug UART port */
    result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,
                                 CY_RETARGET_IO_BAUDRATE);

    /* retarget-io init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Print message */
    /* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */
    printf("\x1b[2J\x1b[;H");
```

Code Listing 1 **Example of SMPU configuration**

```
    printf("------------------------------------------------------------
\r\n");
    printf("XMC7000 MCU: Protection Unit - Shared Memory Protection
(SMPU)\r\n");
    printf("------------------------------------------------------------
\r\n\n");



#if (CY_CPU_CORTEX_M0P)
    printf("[CM0P] Prot Application Main...\r\n");
#else
    /* Disables, cleans and invalidates data cache to avoid write allocate.
*/
    SCB_DisableDCache();
    printf("[CM7] Prot Application Main...\r\n");
#endif /* (CY_CPU_CORTEX_M0P) */

    /**************************/
    /* 1. Setting for MSx_CTL */
    /**************************/
    /* 1.1 Setting for MSx_CTL (for this CPU) to allow the PC value to
become "TP_PROT_CONTEXT" */
    printf("------------------------------------------------------------
\r\n");
    printf("Setting for MSx_CTL (for this CPU) to allow the PC value to
become TP_PROT_CONTEXT \r\n");
    status = Cy_Prot_ConfigBusMaster(MASTER_ID_OF_THIS_CPU, TP_PRIVIREGED,
TP_SECURE, 1 << (TP_PROT_CONTEXT-1));
    CY_ASSERT(status == CY_PROT_SUCCESS);

    /**************************/
    /* 2. Setting for MPU PC  */
    /**************************/
    /* 2.1 Setting for MPU so that this CPU's PC value becomes
"TP_PROT_CONTEXT" */
    printf("Setting for MPU so that this CPU's PC value becomes
TP_PROT_CONTEXT \r\n");
    status = Cy_Prot_SetActivePC(MASTER_ID_OF_THIS_CPU, TP_PROT_CONTEXT);
    CY_ASSERT(status == CY_PROT_SUCCESS);
```

Code Listing 1 **Example of SMPU configuration**

```
    /********************************/
    /* 3. Setting for SMPU_STRUCT 2 */
    /********************************/
    printf("Setting for SMPU_STRUCT 2, memory address: 0x%lx  \r\n", 
TP_PERMITTED_ADDR);

    /* 3.1 Disable SMPU_STRUCT 2 */
    status = Cy_Prot_DisableSmpuSlaveStruct(PROT_SMPU_SMPU_STRUCT2);
    CY_ASSERT(status == CY_PROT_SUCCESS);

    /* 3.2 Setting SMPU_STRUCT 2 for PERMITTED area */
    status = Cy_Prot_ConfigSmpuSlaveStruct(PROT_SMPU_SMPU_STRUCT2, 
&smpuStruct2Config);
    CY_ASSERT(status == CY_PROT_SUCCESS);

    /* 3.3 Enable SMPU_STRUCT 2 */
    status = Cy_Prot_EnableSmpuSlaveStruct(PROT_SMPU_SMPU_STRUCT2);
    CY_ASSERT(status == CY_PROT_SUCCESS);

    /********************************/
    /* 4. Setting for SMPU_STRUCT 3 */
    /********************************/
    printf("Setting for SMPU_STRUCT 3, memory address: 0x%lx \r\n", 
TP_PROHIBITED_ADDR);

    /* 4.1 Disable SMPU_STRUCT 3 */
    status = Cy_Prot_DisableSmpuSlaveStruct(PROT_SMPU_SMPU_STRUCT3);
    CY_ASSERT(status == CY_PROT_SUCCESS);

    /* 4.2 Setting SMPU_STRUCT 3 for PROHIBITED area */
    status = Cy_Prot_ConfigSmpuSlaveStruct(PROT_SMPU_SMPU_STRUCT3, 
&smpuStruct3Config);
    CY_ASSERT(status == CY_PROT_SUCCESS);

    /* 4.3 Enable SMPU_STRUCT 3 */
    status = Cy_Prot_EnableSmpuSlaveStruct(PROT_SMPU_SMPU_STRUCT3);
    CY_ASSERT(status == CY_PROT_SUCCESS);

    /********************************/
```

Code Listing 1 **Example of SMPU configuration**

```
    /* 5.    Memory access test1    */
    /*********************************/
    /* 5.1 This area can be accessed by this master */
    printf("\r\n");
    printf("------------------------------------------------------------
\r\n");
    printf ("This area 0x%lx can be accessed by this master \r\n",
TP_PERMITTED_ADDR);
    printf("Writing 0xA5A5A5A5 to Permitted address 0x%lx \r\n",
TP_PERMITTED_ADDR);
    *(uint32_t*)(TP_PERMITTED_ADDR) = 0xA5A5A5A5; /* write dummy data */
    //*(volatile uint32_t*)(TP_PERMITTED_ADDR);     /* dummy read */
    printf("Reading 0x%lx from address 0x%lx \r\n", *(volatile
uint32_t*)(TP_PERMITTED_ADDR), TP_PERMITTED_ADDR);

    /*********************************/
    /* 6.    Memory access test2    */
    /*********************************/
    /* 6.1 This area can NOT be accessed by this master */
    /* These accessing are expected to cause HardFault */
    printf("------------------------------------------------------------
\r\n");
    printf ("This area 0x%lx can NOT be accessed by this master \r\n",
TP_PROHIBITED_ADDR);
    printf("Writing 0xA5A5A5A5 to Prohibited address 0x%lx \r\n",
TP_PROHIBITED_ADDR);
    printf ("write dummy data <- will be get HardFault here \r\n");
    *(uint32_t*)(TP_PROHIBITED_ADDR) = 0xA5A5A5A5; /* write dummy data <-
will be get HardFault here. */

    //*(volatile uint32_t*)(TP_PROHIBITED_ADDR);     /* dummy read */
    printf("Reading 0x%lx from address 0x%lx \r\n", *(volatile
uint32_t*)(TP_PROHIBITED_ADDR), TP_PROHIBITED_ADDR);

    for (;;)
    {
        Cy_SysLib_Delay(1000);
    }
}
```

Code Listing 1 **Example of SMPU configuration**

```
/***************************************************************************
*****
* Macros
****************************************************************************
****/
#if (CY_CPU_CORTEX_M0P)
    #define MASTER_ID_OF_THIS_CPU CPUSS_MS_ID_CM0
#else
    #define MASTER_ID_OF_THIS_CPU CPUSS_MS_ID_CM7_0
#endif /* (CY_CPU_CORTEX_M0P) */
```

Code Listing 2 Cy_Prot_ConfigBusMaster() function

```
cy_en_prot_status_t Cy_Prot_ConfigBusMaster(en_prot_master_t busMaster,
bool privileged, bool secure, uint32_t pcMask)
{
    cy_en_prot_status_t status = CY_PROT_SUCCESS;
    uint32_t regVal;
    volatile uint32_t *addrMsCtl; /* addrMsCtl is pointer to a register
that is volatile by
                                   * nature as can be changed outside of
firmware control.
                                   */

    CY_ASSERT_L1(CY_PROT_IS_BUS_MASTER_VALID(busMaster));
    CY_ASSERT_L2(CY_PROT_IS_PC_MASK_VALID(pcMask));

    /* Get the address of Master x protection context control register
(MSx_CTL) */
    addrMsCtl = (uint32_t *)(CY_PROT_BASE + (uint32_t)((uint32_t)busMaster
<< CY_PROT_MSX_CTL_SHIFT));

    /* Get bitfields for MSx_CTL */
    regVal = _VAL2FLD(PROT_SMPU_MS0_CTL_NS, !secure)    /* Security setting
*/
           | _VAL2FLD(PROT_SMPU_MS0_CTL_P, privileged) /* Privileged
setting */
           | _VAL2FLD(PROT_SMPU_MS0_CTL_PC_MASK_15_TO_1, pcMask); /*
Protection context mask */
```

Code Listing 2 Cy_Prot_ConfigBusMaster() function

```
    /* Set the value of MSx_CTL */
    *addrMsCtl = regVal;

    /* Check if the MSx_CTL register is successfully updated with the new
register value.
     * The register will not be updated for the invalid master-protection
context.
     */
    status = (*addrMsCtl != regVal) ? CY_PROT_FAILURE : CY_PROT_SUCCESS;

    return status;
}
```

Note: *(*) This process This process specifies the value of the protection context that can be set by the corresponding master. In secure system, it is run by secure muster. See "Protection Properties of Bus Transfer" for more details.*

Code Listing 3 Cy_Prot_DisableSmpuSlaveStruct() function

```
cy_en_prot_status_t
Cy_Prot_DisableSmpuSlaveStruct(PROT_SMPU_SMPU_STRUCT_Type* base)
{
    cy_en_prot_status_t status;

    CY_ASSERT_L1(NULL != base);

    PROT_SMPU_SMPU_STRUCT_ATT0(base) &=
~_VAL2FLD(PROT_SMPU_SMPU_STRUCT_ATT0_ENABLED, CY_PROT_STRUCT_ENABLE);
    status = (_FLD2VAL(PROT_SMPU_SMPU_STRUCT_ATT0_ENABLED,
PROT_SMPU_SMPU_STRUCT_ATT0(base)) == CY_PROT_STRUCT_ENABLE) ?
                CY_PROT_FAILURE : CY_PROT_SUCCESS;

    return status;
}
```

Code Listing 4 Cy_Prot_ConfigSmpuSlaveStruct() function

```
cy_en_prot_status_t 
Cy_Prot_ConfigSmpuSlaveStruct(PROT_SMPU_SMPU_STRUCT_Type* base, const 
cy_stc_smpu_cfg_t* config)
{
    cy_en_prot_status_t status = CY_PROT_SUCCESS;
    uint32_t addrReg;
    uint32_t attReg;

    CY_ASSERT_L1(NULL != base);
    CY_ASSERT_L2(CY_PROT_IS_PC_MASK_VALID(config->pcMask));
    CY_ASSERT_L3(CY_PROT_IS_SMPU_SL_PERM_VALID(config->userPermission));
    CY_ASSERT_L3(CY_PROT_IS_SMPU_SL_PERM_VALID(config->privPermission));
    CY_ASSERT_L3(CY_PROT_IS_REGION_SIZE_VALID(config->regionSize));

    if(((uint32_t)config->pcMask & CY_PROT_SMPU_PC_LIMIT_MASK) != 0UL)
    {
        /* PC mask out of range - not supported in device */
        status = CY_PROT_BAD_PARAM;
    }
    else
    {
        addrReg= _VAL2FLD(PROT_SMPU_SMPU_STRUCT_ADDR0_SUBREGION_DISABLE, 
config->subregions)
                        | _VAL2FLD(PROT_SMPU_SMPU_STRUCT_ADDR0_ADDR24, 
(uint32_t)((uint32_t)config->address >> CY_PROT_ADDR_SHIFT));
        attReg= ((uint32_t)config->userPermission & 
CY_PROT_ATT_PERMISSION_MASK)
                        | (uint32_t)(((uint32_t)config->privPermission & 
CY_PROT_ATT_PERMISSION_MASK) << CY_PROT_ATT_PRIV_PERMISSION_SHIFT)
                        | _VAL2FLD(PROT_SMPU_SMPU_STRUCT_ATT0_NS, !(config-
>secure))
                        | _VAL2FLD(PROT_SMPU_SMPU_STRUCT_ATT0_PC_MASK_15_TO_1, 
config->pcMask)
                        | _VAL2FLD(PROT_SMPU_SMPU_STRUCT_ATT0_REGION_SIZE, 
config->regionSize)
                        | _VAL2FLD(PROT_SMPU_SMPU_STRUCT_ATT0_PC_MATCH, 
config->pcMatch);
        PROT_SMPU_SMPU_STRUCT_ATT0(base) = attReg;
        PROT_SMPU_SMPU_STRUCT_ADDR0(base) = addrReg;
```

Code Listing 4 Cy_Prot_ConfigSmpuSlaveStruct() function

```
        status = ((PROT_SMPU_SMPU_STRUCT_ADDR0(base) != addrReg) ||
((PROT_SMPU_SMPU_STRUCT_ATT0(base) & CY_PROT_SMPU_ATT0_MASK) != attReg))
            ? CY_PROT_FAILURE : CY_PROT_SUCCESS;
    }

    return status;
}
```

Code Listing 5 Cy_Prot_EnableSmpuSlaveStruct() function

```

cy_en_prot_status_t
Cy_Prot_EnableSmpuSlaveStruct(PROT_SMPU_SMPU_STRUCT_Type* base)
{
    cy_en_prot_status_t status;

    CY_ASSERT_L1(NULL != base);

    PROT_SMPU_SMPU_STRUCT_ATT0(base) |=
_VAL2FLD(PROT_SMPU_SMPU_STRUCT_ATT0_ENABLED, CY_PROT_STRUCT_ENABLE);
    status = (_FLD2VAL(PROT_SMPU_SMPU_STRUCT_ATT0_ENABLED,
PROT_SMPU_SMPU_STRUCT_ATT0(base)) != CY_PROT_STRUCT_ENABLE) ?
                CY_PROT_FAILURE : CY_PROT_SUCCESS;

    return status;
}
```

5.4 Configuration example of PPU

The PPU provides the peripheral protection function and is shared by all bus masters.

Generally, the peripheral register address is fixed. Therefore, there are two types of PPUs: Fixed PPU with fixed protected address range, and Programmable PPU with programmable protected address range. Typically, the protection base address and size of the programmable PPU are set by CM0+ with the Protection Context 0 in the boot process.

This section explains how to configure a fixed PPU. A fixed PPU is the same as a programmable PPU except that the protected address area is fixed.

A PPU has also protection pair structure with master/slave settings. Therefore, the setting attribute of slave structure is restricted by the master structure setting.

5.4.1 Usage assumptions

The PPU distinguishes User/Privileged, Secure/Non-Secure, and Protection Context. Fixed PPU configuration is described according to the following use cases.

- Used fixed PPU 228 (GPIO_ENH Port #0)
- Protection context = 5: Privileged and User are not allowed to access to GPIO Port#0.
- Protection context = 6: Privileged and User are allowed to access to GPIO Port#0.

Table 10 shows an example of access restriction for this fixed PPU.

Table 10 Example access restriction for PPU

Setting PPU	Protection context	Privileged	User	Secure
PPU_FX228	PC = 5	No Access	No Access	Non-Secure
	PC = 6	Read/Write	Read/Write	

PPU_FX0 is allowed only CM0+-privileged access. In addition, accesses need the secured attribute. The application software of CM7 is not allowed access this area.

PPU_FX1 is allowed only CM7 access. It is an area dedicated to the application software.

PPU_FX2 is an area where only CM0+ can access. CM7 is allowed read-only access with privileged access.

PPU_FX3 is an area where both CM0+ and CM7 can access with privileged access.

Note: See the datasheet [1] for actual addresses and peripheral channel numbers of target product.

5.4.2 Setting procedure for PPU

Figure 15 shows an example of the setting procedure.

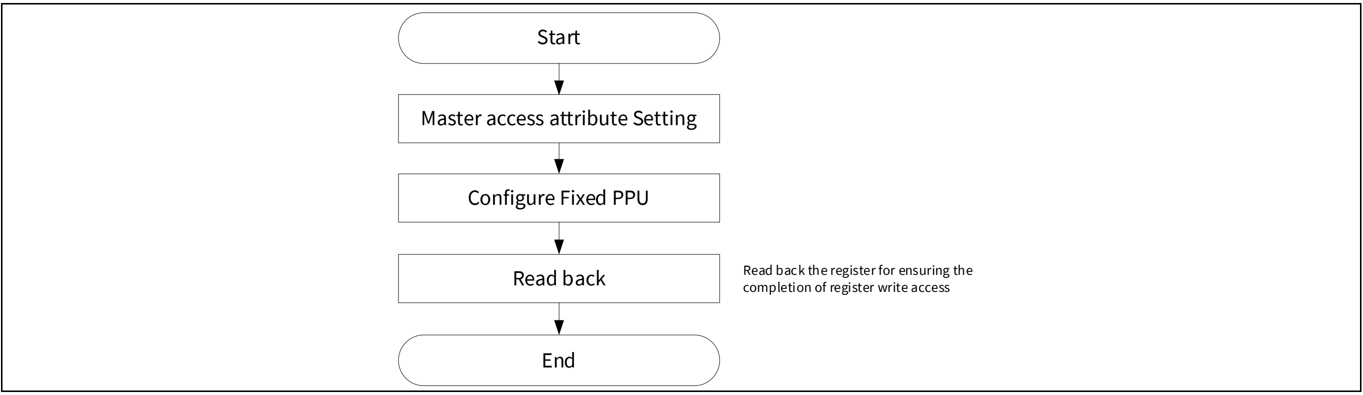
Start
Master access attribute Setting
Configure Fixed PPU
Read back
Read back the register for ensuring the completion of register write access
End

Figure 15 Setting procedure for example of the PPU

The access attribute for the slave structure (PROT_PPU_FXx_SL_ATT0) setting is allowed by the master structure (PROT_PPU_FXx_MS_ATT0).

It is necessary to read back the register for ensuring the completion of register write access when SMPU setting is completed.

5.4.3 Configuration

Table 11 and Table 12 list the parameters and functions of the configuration part in PDL for fixed PPU configuration.

Table 11 Parameters

Parameters	Description	Value
MASTER_ID_OF_THIS_CPU	Defines the master for which PROT_SMPU_MSx_CTL is set.	CPUSS_MS_ID_CM0 (CM0+)
PERI_MS_PPU_FX_GPIO_PRT0_PRT	Defines the configure fixed PPU register address of target	(volatile stc_PERI_MS_PPU_FX_t*) &PERI_MS->PPU_FX[228] (Shows the base address of Fixed PPU 228)
TP_PRIVILEGED	Defines the PROT_SMPU_MSx_CTL.P value	1ul (Privileged mode)
TP_SECURE	Defines the PROT_SMPU_MSx_CTL.NS value	0ul (Non-Secure)
TP_PERMITTED_PC	Defines the protection context for permitted access	CY_PROT_PC6 (6ul)
TP_PROHIBITED_PC	Defines the protection context for not permitted access	CY_PROT_PC5 (5ul)
TP_PERMITTED_PC_MASK	Defines the PROT_SMPU_MSx_CTL.PC_MASK value for the bus master	1ul << (TP_PERMITTED_PC-1ul)

Parameters	Description	Value
TP_PROHIBITED_PC_MASK	Defines the PROT_SMPU_MSx_CTL.PC_MASK value for bus master	1ul << (TP_PROHIBITED_PC-1ul)
ppuFixedAttr_AllEnable.userPermission	Sets the user access attribute for the target PC of Fixed PPU	CY_PROT_PERM_RWX (Full access)
ppuFixedAttr_AllEnable.privPermission	Sets the privileged access attribute for target PC of Fixed PPU	CY_PROT_PERM_RWX (Full access)
ppuFixedAttr_AllEnable.secure	Sets the non-secured access attribute for the target PC of the fixed PPU	TP_SECURE (Non-Secure)
ppuFixedAttr_AllDisable.userPermission	Sets the User access attribute for the target PC of the fixed PPU	CY_PROT_PERM_DISABLED (No Access)
ppuFixedAttr_AllDisable.privPermission	Sets the Privileged access attribute for the target PC of the fixed PPU	CY_PROT_PERM_DISABLED (No Access)
ppuFixedAttr_AllDisable.secure	Sets the non-secured access attribute for the target PC of the fixed PPU	TP_SECURE (Non-Secure)

Table 12 Functions

Functions	Description	Value
`Cy_Prot_ConfigPpuFixedSlaveStruct(*base, setPC, config)`	Configures the Fixed PPU register *base; Base address of the fixed PPU structure setPC: Indicate setting PC config: Access attribute	*base: PERI_MS_PPU_FX_GPIO_PRT0_PRT setPC: TP_PERMITTED_PC or TP_PROHIBITED_PC config: &ppuFixedAttr_AllEnable or &ppuFixedAttr_AllDisable

Code Listing 6 shows an example of the fixed PPU configuration.

Code Listing 6 **Example of fixed PPU configuration**

```
/***************************************************************************
*****
* Macros
***************************************************************************
****/
#define PERI_MS_PPU_FX_GPIO_PRT0_PRT            ((PERI_MS_PPU_FX_Type*)
&PERI_MS->PPU_FX[352])


#if (CY_CPU_CORTEX_M0P)
    #define MASTER_ID_TO_BE_CHECKED CPUSS_MS_ID_CM0
#else
```

Code Listing 6 **Example of fixed PPU configuration**

```
    #define MASTER_ID_TO_BE_CHECKED CPUSS_MS_ID_CM7_0
#endif /* (CY_CPU_CORTEX_M0P) */

#define TP_PRIVIREGED          (1)              /* privileged */
#define TP_SECURE              (0)              /* non secure */
#define TP_PERMITTED_PC        (CY_PROT_PC6)    /* context 6 */
#define TP_PROHIBITED_PC       (CY_PROT_PC5)    /* context 5 */
#define TP_PERMITTED_PC_MASK   (CY_PROT_PCMASK6) //(1 << (TP_PERMITTED_PC-
1u))
#define TP_PROHIBITED_PC_MASK (CY_PROT_PCMASK5) //(1 << (TP_PROHIBITED_PC-
1u))

int main(void)
{
    /* Variable to capture return value of functions */
      cy_rslt_t result = CY_RSLT_SUCCESS;

      cy_en_prot_status_t status;

    /* Initialize the device and board peripherals */
    result = cybsp_init();

    /* Board init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Enable global interrupts */
    __enable_irq();

    /* Initialize retarget-io to use the debug UART port */
    result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,
                                 CY_RETARGET_IO_BAUDRATE);

    /* retarget-io init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
```

Code Listing 6 **Example of fixed PPU configuration**

```
        CY_ASSERT(0);
    }

    /* Print message */
    /* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */
    printf("\x1b[2J\x1b[;H");
    printf("-------------------------------------------------------------
\r\n");
    printf("XMC7000 MCU: Protection Unit - Peripheral Protection
(PPU)\r\n");
    printf("-------------------------------------------------------------
\r\n\n");



#if (CY_CPU_CORTEX_M0P)
    printf("[CM0P] Prot Application Main...\r\n");
#else
    /* Disables, cleans and invalidates data cache to avoid write allocate.
*/
    SCB_DisableDCache();
    printf("[CM7] Prot Application Main...\r\n");
#endif /* (CY_CPU_CORTEX_M0P) */

    /********************************************/
    /* 1. Setting for GPIO 0 Register attribute */
    /********************************************/
    printf("Configure PC5 denied to access GPIO P0 \r\n");
    /* 1_1. Set permissions so that master whose PC is 5 can not access
GPIO 0 */
    status = Cy_Prot_ConfigPpuFixedSlaveAtt(PERI_MS_PPU_FX_GPIO_PRT0_PRT,
TP_PROHIBITED_PC_MASK, CY_PROT_PERM_DISABLED, CY_PROT_PERM_DISABLED,
TP_SECURE);
    CY_ASSERT(status == CY_PROT_SUCCESS);

    printf("Configure PC6 allowed to access GPIO P0 \r\n");
    /* 1_2. Set permissions so that master whose PC is 6 can access GPIO 0
*/
    status = Cy_Prot_ConfigPpuFixedSlaveAtt(PERI_MS_PPU_FX_GPIO_PRT0_PRT,
TP_PERMITTED_PC_MASK, CY_PROT_PERM_RW, CY_PROT_PERM_RW, TP_SECURE);
    CY_ASSERT(status == CY_PROT_SUCCESS);
```

Code Listing 6 **Example of fixed PPU configuration**

```
    /***********************************************/
    /* 2. Setting for MPUx (MPU for this Master) */
    /***********************************************/
    printf("Setting for MSx_CTL to allow the PC value to become PC6 or PC5 
\r\n");
    /* 2_1. Setting for MSx_CTL to allow the PC value to become 
"TP_PERMITTED_PC" or "TP_PROHIBITED_PC" */
    status = Cy_Prot_ConfigBusMaster(MASTER_ID_TO_BE_CHECKED, 
TP_PRIVIREGED, TP_SECURE, TP_PERMITTED_PC_MASK|TP_PROHIBITED_PC_MASK);
    CY_ASSERT(status == CY_PROT_SUCCESS);

    /*******************************/
    /* 3. Access with PC permitted */
    /*******************************/
    printf("Setting Permitted PC %d to bus master %d \r\n", 
TP_PERMITTED_PC, MASTER_ID_TO_BE_CHECKED);
    /* 3_1. Setting for MPU so that this CPU's PC value becomes 
"TP_PERMITTED_PC" */
    status = Cy_Prot_SetActivePC(MASTER_ID_TO_BE_CHECKED, TP_PERMITTED_PC);
    CY_ASSERT(status == CY_PROT_SUCCESS);

    /* 3_2. Read from GPIO 0 register */
    printf ("Reading GPIO P0_OUT register 0x%lx \r\n", GPIO_PRT0->OUT);



    /********************************/
    /* 4. Access with PC prohibited */
    /********************************/
    printf("Setting Prohibited PC %d to bus master %d \r\n", 
TP_PROHIBITED_PC, MASTER_ID_TO_BE_CHECKED);
    /* 4_1. Setting for MPU so that this CPU's PC value becomes 
"TP_PROHIBITED_PC" */
    status = Cy_Prot_SetActivePC(MASTER_ID_TO_BE_CHECKED, 
TP_PROHIBITED_PC);
    CY_ASSERT(status == CY_PROT_SUCCESS);

    /* 4_2. Read from GPIO 0 register (this will cause hardfault) */
    printf ("Read from GPIO 0 register (this will cause hardfault) \r\n");
    printf ("Reading GPIO P0_OUT register 0x%lx \r\n", GPIO_PRT0->OUT);
```

Code Listing 6 **Example of fixed PPU configuration**

```
    for (;;)
    {
        Cy_SysLib_Delay(1000);
    }
}
```

Code Listing 7 **Cy_Prot_ConfigPpuFixedSlaveStruct() function**

```
cy_en_prot_status_t Cy_Prot_ConfigPpuFixedSlaveAtt(PERI_MS_PPU_FX_Type* 
base, uint16_t pcMask,
                           cy_en_prot_perm_t userPermission, 
cy_en_prot_perm_t privPermission, bool secure)
{
    /* The parameter checks */
    CY_ASSERT_L1(NULL != base);
    CY_ASSERT_L3(CY_PROT_IS_FIXED_MS_PERM_VALID(userPermission));
    CY_ASSERT_L3(CY_PROT_IS_FIXED_MS_PERM_VALID(privPermission));

    CY_MISRA_DEVIATE_LINE('MISRA C-2012 Rule 18.1','Checked manually, base 
pointer will not exceed register range.');
    return (Prot_ConfigPpuAtt(PERI_MS_PPU_FX_SL_ATT(base), pcMask, 
userPermission, privPermission, secure));
}
```

6 Glossary

Terms	Description
MPU	Memory Protection Unit
SMPU	Shared Memory Protection Unit
PPU	Peripheral Protection Unit
PCs	Protection Contexts. See the “Protection Context” section in the Direct Memory Access chapter of the architecture reference manual for details.
DMB	Data memory barrier. It is instruction in the Thumb instruction set.
DSB	Data Synchronization Barrier. It is instruction in the Thumb instruction set.
ISB	Instruction Synchronization Barrier. It is instruction in the Thumb instruction set.
P-DMA (DW)	Peripheral DMA. See the Direct Memory Access chapter of the architecture reference manual for details.
M-DMA (DMAC)	Memory DMA. See the Direct Memory Access chapter of the architecture reference manual for details.
CRYPTO	Cryptography Component. See the Cryptography Block chapter of the architecture reference manual for details.

References

The following are the XMC7000 datasheets, technical reference manuals, and application notes. Contact Technical Support to obtain XMC7000 family reference documents and PDL.

[1] Device datasheet

- XMC7100 series 32-bit Arm® Cortex®-M7 microcontroller datasheet
- XMC7200 series 32-bit Arm® Cortex®-M7 microcontroller datasheet

[2] Reference manuals

- XMC7000 MCU family architecture reference manual
- XMC7100 register reference manual
- XMC7200 register reference manual

[3] Application notes

- AN234802 - Secure system configuration in XMC7000 family
- AN234334 - Getting started with XMC7000 MCU on ModusToolbox™
- AN234254 - Multi-core handling guide in XMC7000
- AN234196 - Using the CRYPTO module in XMC7000 family

Revision history

Document revision	Date	Description of changes
**	2022-01-25	Initial release
*A	2022-04-28	Updated code snippets to follow the PDL (3.0) version Changed m4cpuss to m7cpuss in Figure 1 Updated XMC7100 and added XMC7200 bus master ID in Table 4 Updated SRAM address from 0x08000000 to 0x28000000
*B	2024-03-19	Updated references documents Added description in 3.1 Protection properties of bus transfer. Added Note in 3.5.2 PC_MATCH operation. Updated example code

Edition 2024-03-19

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

002-34279 Rev.*B