Application Note 296

Managing Memory Protection Performance Concerns in Cached Cortex^{TM} R4/R5 Processors

Document number: ARM DAI 296A

Issued: 20th July, 2011

Copyright ARM Limited 2011

Application Note 296
Managing Memory Protection Performance Concerns in Cached Cortex R4/R5 Processor

Release information

Change history

Date	**Issue**	**Change**
July 2011	A	First release
		2nd release
		3rd release

Proprietary notice

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

`http://www.arm.com`

Table of Contents

1. Introduction

This document provides guidance for programming the Memory Protection Unit (MPU) in the ARM Cortex-R4 and Cortex-R5 processors when the protection configuration is changed during a context switch. Programmers might assume that all caches must be flushed whenever the MPU is reprogrammed. However, this is rarely the case and doing so may significantly degrade system performance.

1.1 Scope

This document assumes familiarity with the ARM Cortex-R4 and Cortex-R5 processors and the ARMv7-R real-time architecture. It is intended for C and assembler programmers to enhance their understanding of the MPU behavior.

1.2 Additional reading

The following documents published by ARM contain information relevant to this document:

- Cortex-R4 and Cortex-R4F Technical Reference Manual (ARM DDI 0363)
- Cortex-R5 and Cortex-R5F Technical Reference Manual (ARM DDI 0460)
- ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition (ARM DDI 0406)

ARM documentation is accessed through the Infocenter at http://infocenter.arm.com.

2. Memory Protection Unit Basics

Many embedded systems operate with a multitasking operating system which provides a facility to ensure that the task currently executing does not disrupt the operation of other tasks. System resources and the code and data of other tasks are protected. The protection system typically relies on both hardware and software to do this. In a system with no hardware protection support, each task must work in a cooperative way with other tasks and follow rules.

In contrast, a system with dedicated protection hardware will check and restrict access to system resources, preventing hostile or unintentional access to forbidden resources. Tasks are still required to follow a set of OS rules, but these are also enforced by hardware, which gives more robust protection.

ARM provides many processors with this capability, using either a memory protection unit (MPU) or a memory management unit (MMU). This Applications Note is about MPU based processors. These provide hardware protection over a number of software-programmed regions, but stop short of providing a full virtual memory system with address translation, which requires an MMU.

An ARM MPU uses regions to manage system protection. A *region* is a set of attributes associated with an area of memory. The processor core holds these attributes in CP15 registers and identifies each region with a number. A region’s memory boundaries are defined by its base address and its size. Each region possesses additional attributes which define access rights, memory type and the cache policies. Because peripherals are memory-mapped in ARM systems, the same protection mechanism is used for both system peripherals and task memory.

In the Cortex-R4 and Cortex-R5 processors, the presence of the MPU is optional although generally included. If present, there may be either 8, 12 or 16 such regions (defined by the hardware implementer at RTL configuration stage). The smallest length (size) of a region is just 32 bytes. If a region is of 256 bytes or more, it may be divided into 8 sub-regions. Although the Cortex-R4 and Cortex-R5 processors have a Harvard view of memory, the regions are common to both instruction and data accesses. However, it is possible to use the “Execute Never (XN)” attribute to disallow instructions execution from a peripheral or data region.

2.1 Regions

Each region consists of:

- Base address
- Region Size and Enable
- Memory Type and Access Control
- Optionally sub-regions

The Base address of a region must always be aligned to the region size. This means, for example, that a 32kB region must have a base address aligned to a 32kB boundary.

Initialization and further programming of the MPU regions is achieved through CP15 MCR instructions. Section 2.4 of this Application Note provides a brief description of the CP15 registers – more detail may be found in the Cortex-R4 and Cortex-R5 Technical Reference Manuals and in the ARM Architecture Reference Manual.

None of the regions are defined or enabled after reset. Any access which lies outside a defined and enabled region when the MPU is enabled will cause an abort. Therefore, at least one region must be defined before enabling the MPU afte reset. If the MPU is enabled and no regions are defined, the processor enters a state from which it is recoverable only by a further reset.

However, when the MPU is disabled, the Cortex-R4 and Cortex-R5 processors make use of a default memory map and Privileged accesses can fall through to this. Table 7.1 of the Technical Reference Manual shows this default memory map.

The code which enables the MPU (by setting the BR bit in the Control Register) must be in a memory location which is defined as executable, otherwise the core will immediately take an abort exception when the MPU is enabled.

The available memory types are Normal, Device and Strongly Ordered. A thorough description of these is outside of the scope of this document (but may be found in the ARM Architecture Reference Manual). Similarly, for Normal memory, there a number of possible cache policies (write-back, write-through etc.) which can be selected for a region (see ARM Architecture Reference Manual). This means that, for example, one region may be marked as using write-back cache policy, while another is non-cacheable.

For the purposes of memory protection, it is the Access Control settings which are of interest. Access to a region in memory may be set as read-write, read-only, or no access. It is further qualified by the current processor mode, which may be either privileged or non-privileged (user).

When the processor accesses a memory address, the MPU determines which region's attributes apply to that address and compares the region's access permission attributes with the current processor mode to determine what action is required. If the access is permitted by the region access criteria, the read or write to main memory (or TCM) occurs. If it is not permitted, the access to main memory does not occur and an abort will be generated. This will be either a prefetch or data abort, and the appropriate abort handler will be called.

2.2 Implementing a Protected Memory System with Regions

To implement a protected system, the OS must define a number of regions to cover the different areas in the main memory map. This may be done as a static (fixed) scheme, during the boot sequence, which persists while the system is running. Alternatively, more complex systems may assign (and remove) regions dynamically as tasks start and finish, i.e. as the software context switches.

There are a number of points to consider when dealing with regions:

- Regions are assigned a priority number that is independent of the privilege assigned to the region.
- Regions can overlap other regions. In this case, the attributes of the region with the highest priority number take precedence over the other regions (only for the addresses within the areas that overlap.)
- A region's size can be any power of two between 32B and 4 GB.
- Accessing an area of main memory outside of a defined region results in an abort.

Overlapping regions provide a greater flexibility when assigning access permission. A useful feature provided by overlapping regions is a background region. A low priority region is used to assign the same attributes to a large memory area. Other regions with higher priority are then placed over this background region to change the attributes of a smaller part of the memory map.

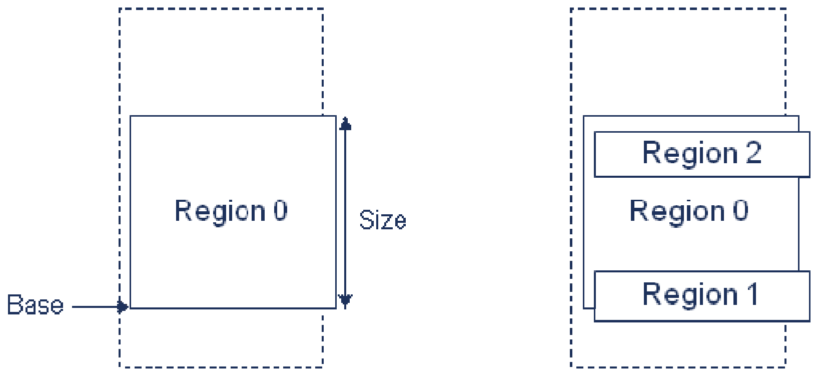

Figure 1: A single region and overlapping regions

For example, if an embedded system defines a large low priority region with privileged access only, it can then overlay a smaller region with user mode access permitted over this background. The location of the smaller region can be moved over different areas of the background region upon context switches to give a number of different user task-specific spaces. Each time that the smaller user-accessible region is moved, the previously covered area becomes protected by the background region. This means that we can provide an area of memory specific to each task, which is protected from all other tasks, with only one or two instructions, to write to the appropriate CP15 register.

It is common practice to define a region which covers the entire 4GB physical address map, which has lower priority than any other region and which provides the memory attributes for accesses which do not match any of the defined memory regions. If it desired to have such accesses generate a memory abort, this can be done using the System Control Register BR bit, which provides this behavior without the need to program region 0 to do so. Alternatively, the default memory map may be used to define the background region for privileged accesses.

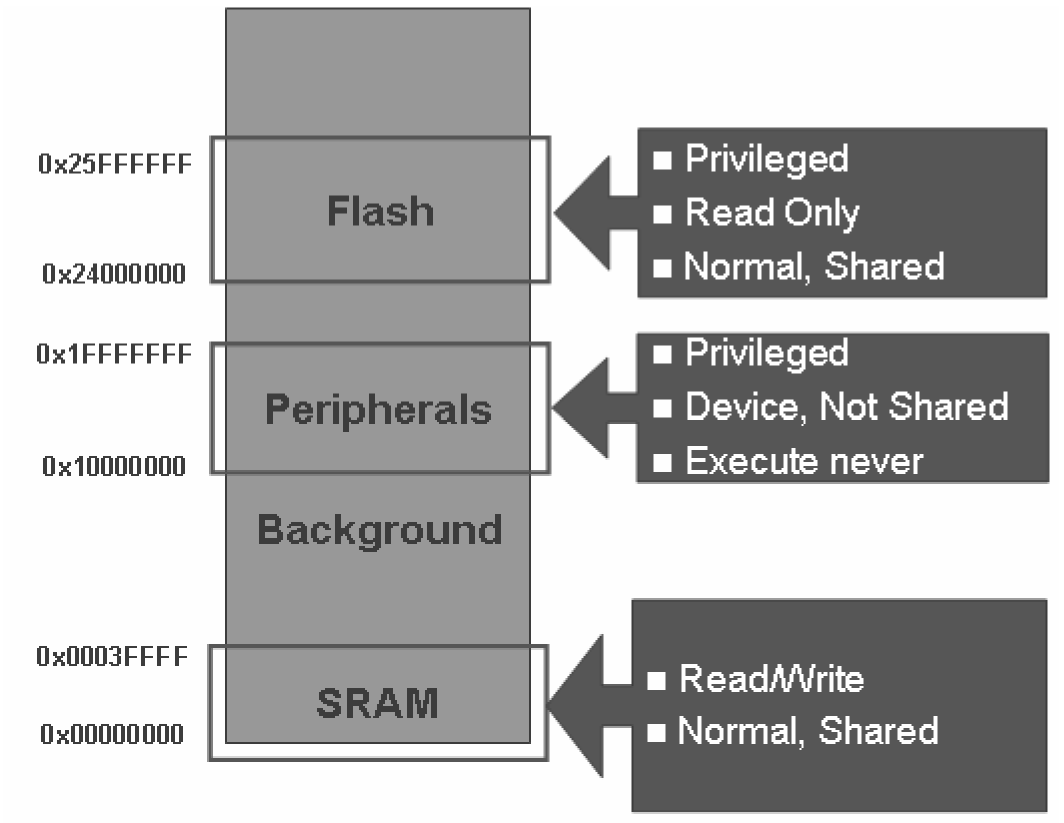

Fig 2 Example Static MPU Region configuration with background region

2.3 Sub-Regions

Each region may be split into eight equal sized non-overlapping sub-regions. This means the granularity of protection and memory attributes can be increased without significant increase in hardware complexity and reduces the number of regions which must be used to align protection boundaries to unaligned addresses. All region sizes between 256 bytes and 4GB support 8 sub-regions, but region sizes below 256 bytes do not support sub-regions.

Each of the sub-regions may be individually enabled or disabled. An access to a memory address in a disabled sub-region does not use the attributes and permissions defined for that region. Instead, it uses the attributes and permissions of a lower priority region or generates a background fault if no other regions overlap at that address. This enables increased protection and memory attribute granularity.

Disabling sub-regions provides backward compatibility with the ARMv6 architecture.

2.4 CP15 Registers

This section provides a brief summary of the CP15 registers and instructions used to control the MPU.

- Memory Region Register Number (MCR p15, 0, Rd, c6, c2, 0)
    - Rd[3:0] contains the region number to select
    - Controls which region number the following accesses apply to

- Region Base (MCR p15, 0, Rd, c6, c1, 0)
    - Rd[31:5] contains MSBs of the base address
    - This must be aligned to the size of the region
    - e.g. If size is 1KB, bits [10:5] must be zero

- Region Size and Enable (MCR p15, 0, Rd, c6, c1, 2)
    - Rd[5:1] sets the size
    - Valid values from 5’b00100 (32 bytes) to 5’b11111 (4GB)
    - Rd[0] is the enable bit
    - At least one valid, enabled region is needed prior to enabling the MPU

- Region Access Control (MCR p15, 0, Rd, c6, c1, 4)
    - Rd[12]: Execute Never (XN)
    - Determines if a region of memory is Executable
    - Rd[10:8]: Access Permission (AP)
    - Defines the data access permissions
    - Rd[5:3]: Type Extensions (TEX)
    - Defines the Type Extensions attributes of the region
    - Combined with the S, C and B bits to determine the memory type
    - Rd[2]: Shared (S)
    - Rd[1]: Cacheable (C)
    - Rd[0]: Bufferable (B)

3. Managing the MPU in Context Switches

The following gives additional guidance for application of the Cortex-R4 and Cortex-R5 processors with an MPU which implements the ARM Protected Memory System Architecture (PMSA). More detail on PMSA can be found in the ARM Architecture Reference Manual for ARM v7-A and R processors.

Section 7.4 of both Cortex-R4 and Cortex-R5 Technical Reference Manuals (TRM) describes the MPU and memory system interaction and it states that the L1 caches must be invalidated and the L1 data cache must be cleaned before enabling or disabling the MPU. In the absence of further guidance, programmers might assume that this should also be done whenever the MPU is reprogrammed, for example, on a context switch. However, this is rarely necessary.

3.1 Attributes and cache maintenance

As we have seen, the MPU configuration is programmed via CP15 which is where address regions are configured. For each region you specify:

- Base address
- Size
- Attributes

So, the MPU can be programmed such that any given memory address can appear in more than one region, in which case the highest numbered matching region will set the attributes for that address. This is called ‘resolving the attributes’ for that address and the final attributes for the given address are termed the ‘resolved attributes’.

MPU attributes for the Cortex-R4 and Cortex-R5 processors are as follows:

1. Memory type (Normal, Device, Strongly-Ordered)
2. Cacheability
3. Shareability
4. Permission control (User/Privileged, Read/Write, Executable)

Note that:

- TCM regions are given Normal, Non-Cacheable, Non-shared type attributes, regardless of the above MPU region settings. TCM permission control settings are preserved from the MPU region settings.
- Cortex-R5 Low Latency Peripheral Port regions are given Non-cacheable and Execute Never region attributes regardless of the MPU region setting. The Memory type, Shareability, User/Privileged and Read/Write permission control settings are preserved from the MPU region settings.

Any of the above attributes may be changed without disabling and re-enabling the MPU, so long as care is taken to ensure that the resolved attributes for the code (and its data) performing the MPU update are not changed.

Cache maintenance is only required when any of the first three attributes already resolved for any memory location are changed by reprogramming the MPU. Coherency in both Level-1 Instruction and Data caches and any Level-2 caches (if present) will then need to be restored by performing the following operations:

- I-cache: invalidate all
- D-cache: clean and invalidate all

It is not necessary to disable and re-enable the MPU whilst restoring coherency in this way.

An alternative strategy might be to invalidate by address (or clean and invalidate for the D-cache) for all addresses that used to be cacheable but now are not. However this will not usually result in improved performance.

Write-Through (WT) and Write-Back (WB) are both different cacheability attributes and any change to them will require cache maintenance. e.g. changing from WB to WT is not safe without a clean and invalidate (or at least a clean).

3.2 Context switching

Context switches should only need to make changes to permission control and this attribute change does <u>not</u> require cache maintenance. In this case, the caches continue to operate correctly without any loss of coherency.

A context switch is expected to only change permission control for certain memory locations. This may involve changes to the boundaries of regions but is not expected to involve changing any of the first three attributes listed above for any memory location. Therefore context switching may be performed efficiently with the MPU remaining enabled and without the need for any cache maintenance.

3.3 Permission violation example

Behavior in case of a permission violation is shown by the following example:

Suppose that a Task 1 programs the MPU as follows:

MPU Region 0: Address from 0x00000000 to 0xFFFFFFFF

Described as Normal memory, Enabled, Cacheable using Write-Back

Executable = No. Permission = No Access

MPU Region 1: Address from 0x10000000 to 0x1FFFFFFF

Described as Normal memory, Enabled, Cacheable using Write-Back

Executable = Yes. Permission = User Read/Write.

And part of Region 1 is cached during the task.

A context switch moves to Task 2 in which:

MPU Region 0: Stays the same

MPU Region 1: Is now Disabled and therefore the addresses in this region hit in Region 0 and thus do not have access permission.

Since only permissions have changed there is no requirement to perform cache maintenance when switching from Task 1 and Task 2.

If the processor then attempts to read or write memory at 0x10001000 it will violate the access permission in which case:

- It will generate a Synchronous Data Abort and,
- The cache may be read depending on the implementation, but the data will always be discarded

- The target register will not be updated
- If the address is not in cache the processor will not start a line fill
- Writes will not be issued to the memory system

3.3.1 A note on cache operation

It is worth noting that, having configured the MPU and after switching from Task 1 to Task 2 as described above:

- Any ‘dirty’ cache lines corresponding to Region 1 will still be evicted as normal, despite no longer being mapped.

3.4 Cache Maintenance Recommendations

As we have seen, it will sometimes be necessary to perform cache maintenance operations (clean and/or invalidate) when changes are made to MPU region attributes.

Specifically, for the Cortex-R4 and R5 processors, it is changing from a less restrictive to a more restrictive attribute that will require cache maintenance, e.g. when changing a region’s resolved attribute from cacheable to non-cacheable. Therefore, in the case of Cortex-R4 and R5 it is possible to identify changes to attributes 1 through 3 which would not necessitate cache maintenance.

However, it is strongly recommended that cache is always maintained when changes are made to attributes 1 through 3 in order that programs remain platform independent. Additional implications may also exist in the level-2 memory system which are outside the domain of the processor and are system-specific.

Overall it is strongly recommended practice for any given memory location to have fixed values for attributes 1 through 3, and that these be independent of the currently executing context.

Failure to guarantee this will mean that the OS must explicitly manage the mismatched attributes, which will involve cache maintenance and other considerations. It should be emphasized that in most systems, attribute changes which require cache maintenance (changes to memory type or cacheability) do not typically occur after system start-up.

3.5 Choosing the best cache policy

System designers should evaluate the policy for cache operation which best suits their requirements. If a write-back policy is used then the cache will often have to be cleaned as described above before switching context so that coherency in the memory system is maintained. This can require lots of writes to CP15 registers prior to switching. Alternatively, choosing a write-through policy can reduce system performance and increase power consumption as coherency is maintained on every cache operation, which is sometimes unnecessary. However, this means that the cache is being cleaned continuously and so cache maintenance will typically take less time.

4. Conclusion

Cortex-R4 and R5 programmers need not disable and re-enable the MPU nor perform cache maintenance when changing permission control and region boundaries for a context switch. Avoiding unnecessary cache maintenance operations will increase system performance and efficiency.