NXP Semiconductors
Application Note

Document Number: AN12042
Rev. 1, 12/2017

Using the i.MXRT L1 Cache

1. Introduction

i.MXRT series takes advantage of the ARM Cortex-M7 core with 32K/32K L1 I/D-Cache. This delivers extremely high performance regardless the code is executed from on-chip RAM, external Flash or external memory.

This documentation introduces the basic technology of the cache system that includes the L1 cache, memory types, attributes and MPU (Memory Protection Unit). It guides user on how to use cache to develop applications running in a correct and high-performance way. It does not intend to dig into details of the cache system, for more detailed information, please refer to ARM Cortex-M7 Processor User Guide.

The software used for example in this documentation are based on the i.MXRT1050 SDK release with ARM's CMSIS implementation. The development environment is IAR Embedded Workbench 8.11. The hardware used to verify the example is MIMXRT1050-EVK board.

Contents

2. Overview

This chapter introduces i.MXRT system architecture with cache related parts. It talks about the L1 cache behavior, ARM cortex-M7 defined memory types/attributes and the MPU (Memory Protection Unit) system. This gives an overview of the i.MXRT cache system and how they affect the application use cases.

2.1. i.MXRT system architecture (cache related)

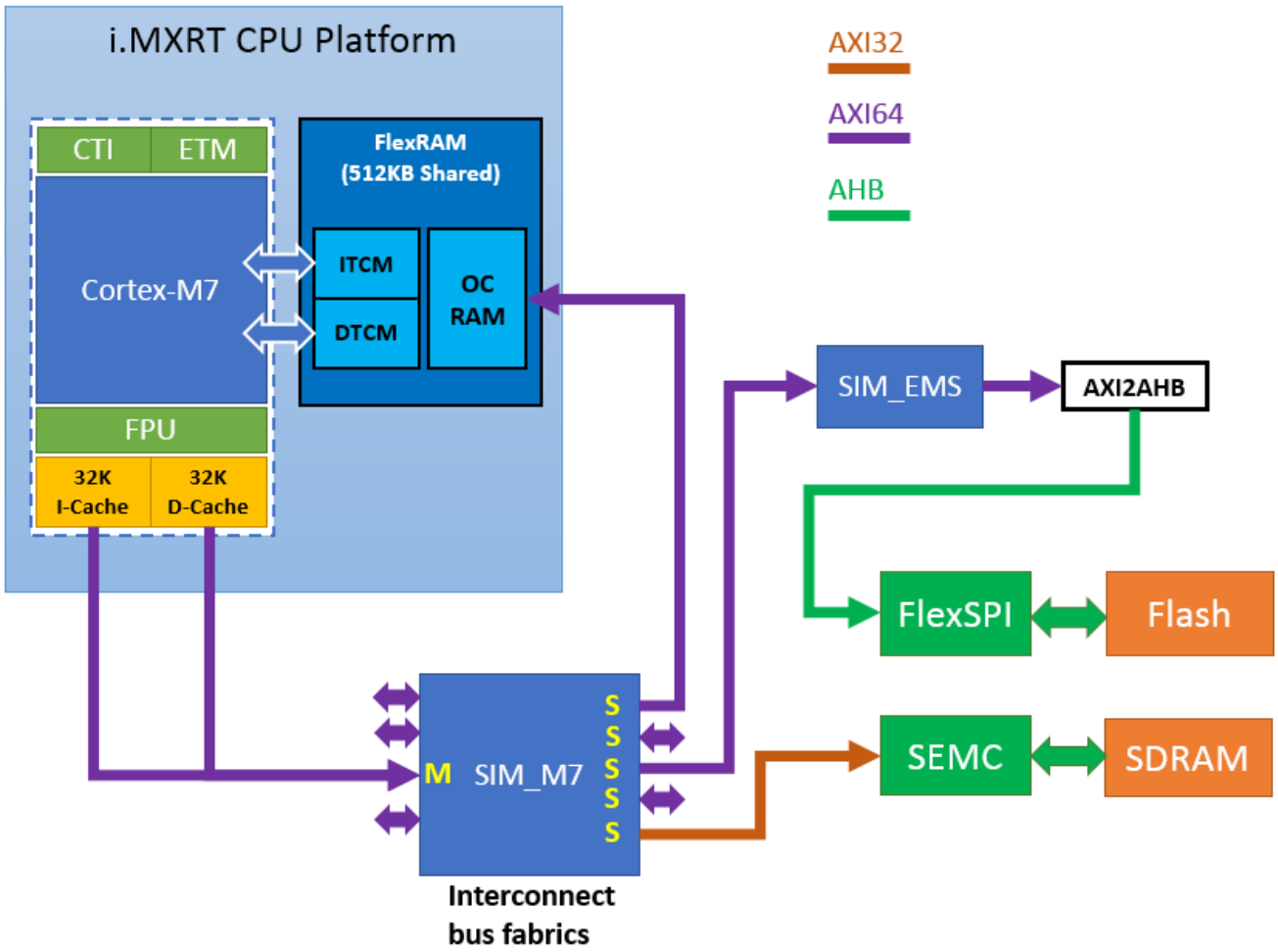

Figure 1. **i.MXRT1050 core and system block diagram**

The i.MXRT series implement a CPU core platform described in Figure 1. The L1 I/D-Cache is embedded in the core platform. The data cache is 4-way set-associative and instruction cache is 2-way set-associative with cache line size of 32 bytes. It connects with the SIM_M7 bus fabric master port by AXI bus. The subsystem of internal/external memory like OCRAM (FlexRAM banks configured as OCRAM), FlexSPI (Serial NOR, NAND Flash and Hyper Flash/RAM etc.) and SEMC (SDRAM,

PNOR Flash, NAND Flash etc.) are connected to the bus fabric slave port. CPU core access the subsystem through this bus fabric by L1 cache.

Since the access to the subsystem of those memory can take multiple cycles (especially on the external memory interfaces with multiple wait states), the L1 cache is designed to speed up the read/write operation to the memory. This brings a big performance boost.

The I/DTCM (FlexRAM banks configured as TCM) is accessed directly by CPU core, bypass the L1 cache. Therefore, put the critical code and data into the TCM is recommended, like the vector table.

2.2. L1 Cache behavior

Any access that is not for a TCM is handled by the appropriate cache controller. If the access is to non-shared cacheable memory, and the cache is enabled, a lookup is performed in the cache and, if found, that is, a cache hit, the data is fetched from or written into the cache. When the cache is not enabled and for non-cacheable or shared memory the accesses are performed using the AXI bus.

Both caches allocate a memory location to a cache line on a cache miss because of a read, that is, all cacheable locations are Read-Allocate. In addition, the data cache can allocate on a write access if the memory location is marked as Write-Allocate. When a cache line is allocated, the appropriate memory is fetched into a linefill buffer by the AXI bus before being written to the cache.

Writes accesses that hit in the data cache are written into the cache RAMs. If the memory location is marked as Write-Through, the write is also performed on the AXI bus, so that the data stored in the RAM remains coherent with the external memory system. If the memory is Write-Back, the cache line is marked as dirty, and the write is only performed on the AXI bus when the line is evicted. When a dirty cache line is evicted, the data is passed to the write buffer in the AXI bus to be written to the external memory system.

2.3. Memory types and attributes

The memory map and the programming of the MPU splits the memory map into regions. Each region has a defined memory type, and some regions have additional memory attributes. The memory type and attributes determine the behavior of accesses to the region.

The memory types are:

- **Normal** – The processor can re-order transactions for efficiency, or perform speculative reads.
- **Device and Strongly-Ordered** – The processor preserves transaction order relative to other transactions to Device or Strongly-Ordered Memory.

The different ordering requirements for Device and Strongly-Ordered Memory mean that the external memory system can buffer a write to Device memory, but must not buffer a write to Strongly-Ordered Memory.

The memory attributes include:

- **Shareable (S) –** 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. For i.MXRT, shareable means **non-cacheable** by default.
- **Execute Never (XN) –** Means that the processor prevents instruction accesses. A fault exception is generated only on execution of an instruction that is executed from an XN region.
- **TEX, Cacheable (C), Bufferable (B) –** Identify the memory type and cache policy used by this region of memory.
- **Access permission (AP) –** access permissions for privileged and unprivileged software. Value can be "No access", RW, RO.

The *memory type, S, TEX, C, B* attributes determine the cache policy that application should take care of. See the next section about the cache policy.

2.3.1. Cache Policy

The TEX/C/B attributes defines the memory type and cache policy applied to the region of memory, here list commonly used combination of these bits:

Table 1. **Cache Policy settings**

TEX	C	B	Memory Type	Cache Policy
0b000	0	0	Strongly Ordered	Non-cacheable
	0	1	Device	Non-cacheable
	1	0	Normal	WT, No WA
	1	1	Normal	WB, No WA
0b001	0	0	Normal	Non-cacheable
	1	1	Normal	WB, WA

Cache policy is fixed to Non-cacheable when Shareable bit is set, no matter what's the TEX/C/B value. A full cache policy settings table can be found in **ARM Cortex-M7 Processor User Guide.**

Each of the cache policy is described here:

- **Write allocation (WA) –** A cache line is allocated on a write miss. This means that executing a store instruction on the processor might cause a burst read to occur.
- **Write-back (WB) –** A write updates the cache only and marks the cache line as dirty. External memory is updated only when the line is evicted or explicitly cleaned.
- **Write-through (WT) –** A write updates both the cache and the external memory system. This does not mark the cache line as dirty.

2.3.2. i.MXRT1050 Memory Map

The default memory map of important regions with memory types and cache policy is listed below *Table 2*. Application can use MPU to configure different memory type and cache policy to overwrite the default.

Table 2. **i.MXRT1050 Memory Map with type and cache policy**

Start	End	Size	Modules	Memory type	Cache Policy
C000_0000	DFFF_FFFF	512MB	SEMC3	Device	Non-Cacheable
A000_0000	BFFF_FFFF	512MB	SEMC2	Device	Non-Cacheable
9000_0000	9FFF_FFFF	256MB	SEMC1	Normal	Cacheable/WT (no WA)
8000_0000	8FFF_FFFF	256MB	SEMC0	Normal	Cacheable/WT (no WA)
7FC0_0000	7FFF_FFFF	4MB	FlexSPI RX FIFO	Normal	Cacheable/WB/WA
7F80_0000	7FBF_FFFF	4MB	FlexSPI TX FIFO	Normal	Cacheable/WB/WA
6000_0000	7F7F_FFFF	504MB	FlexSPI / FlexSPI cipher text	Normal	Cacheable/WB/WA
2020_0000	2027_FFFF	512KB	OCRAM	Normal	Cacheable/WB/WA
2000_0000	2007_FFFF	512KB	DTCM	Normal	-
0000_0000	0007_FFFF	512KB	ITCM	Normal	-

DTCM/ITCM is Tightly-Coupled Memories, core can access it directly (cache is not involved).

Which SEMC memory region used by application is decided by the Chip-Select in the board design. For example, SDRAM use SEMC_CS0, then we access SDRAM by SEMC0 memory region

2.4. MPU (Memory Protection Unit)

The Memory Protection Unit (MPU) divides the memory map into a few regions, and defines the location, size, access permissions, and memory attributes of each region. It supports:

- Independent attribute settings for each region
- Overlapping regions
- Export of memory attributes to the system

The memory attributes affect the behavior of memory accesses to the region. The i.MXRT MPU defines:

- 16 separate memory regions, 0-15
- A background region

When memory regions overlap, a memory access is affected by the attributes of the region with the highest number. For example, the attributes for region 15 take precedence over the attributes of any region that overlaps region 15. The background region has the same memory access attributes as the default memory map, but is accessible from privileged software only.

The MPU memory map is unified. This means instruction accesses and data accesses have same region settings. If a program accesses a memory location that is prohibited by the MPU, the processor generates a MemManage fault. This causes a fault exception, and might cause termination of the process in an OS environment.

Typically, application or embedded OS uses the MPU for memory protection and memory cache policy configurations. Please see the section 4.2 for how to use MPU to configure memory region for different cache policy.

2.5. Hardware L1 I-cache prefetching

The speculative fetch is likely caused by branch prediction in Cortex-M7,when the branch predictor is enabled, the core will attempt to fetch ahead of the current execution point, while the branch predictor is disabled, then the core will still do a small amount of prediction (backwards direct branches will be predicted to be taken, forwards direct branches will be predicted to be not taken), so even when branch prediction is disabled, there's a small chance that the core can start fetching to unexpected locations.

NOTE

Speculative fetch will be performed to Normal Memory, but has no affect to Strongly Ordered or Device memory, and speculative instruction fetches will never be performed to XN memory (refer to *2.3* for memory type).

3. Cache operation

There are three types of cache operations:

- **Cache Enable/Disable –** Cache on/off
- **Cache Clean –** Writes back dirty cache lines to the memory (sometimes called a flush)
- **Cache Invalidate –** Marks the contents in cache as invalid (basically, a delete operation)

i.MXRT SDK provides two ways to do the cache operations

3.1. Accessing the cache using CMSIS

Table 3. **CMSIS cache functions**

CMSIS function	Descriptions
void SCB_EnableICache (void)	Invalidate and then enable instruction cache
void SCB_DisableICache (void)	Disable instruction cache and invalidate its contents
void SCB_InvalidateICache (void)	Invalidate instruction cache
void SCB_EnableDCache (void)	Invalidate and then enable data cache
void SCB_DisableDCache (void)	Disable data cache and then clean and invalidate its contents
void SCB_InvalidateDCache (void)	Invalidate data cache
void SCB_CleanDCache (void)	Clean data cache
void SCB_CleanInvalidateDCache (void)	Clean and invalidate data cache

void SCB_InvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize)	Invalidate data cache by address. @addr must aligned to 32-bytes boundary @dsize in bytes
void SCB_CleanDCache_by_Addr (uint32_t *addr, int32_t dsize)	Clean data cache by address @addr must aligned to 32-bytes boundary @dsize in bytes
void SCB_CleanInvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize)	Clean and invalidate data cache by address @addr must aligned to 32-bytes boundary @dsize in bytes

3.2. Accessing the cache using SDK

i.MXRT SDK provides a cache driver for L1 cache operations, which is a wrapper to the CMSIS cache functions:

```
void L1CACHE_DisableICache(void)
void L1CACHE_InvalidateICache(void)
void L1CACHE_InvalidateICacheByRange(uint32_t address, uint32_t size_byte)
void L1CACHE_EnableDCache(void)
void L1CACHE_DisableDCache(void)
void L1CACHE_InvalidateDCache(void)
void L1CACHE_InvalidateDCacheByRange(uint32_t address, uint32_t size_byte)
void L1CACHE_CleanDCacheByRange(uint32_t address, uint32_t size_byte)
void L1CACHE_CleanInvalidateDCacheByRange(uint32_t address, void uint32_t size_byte)
```

For more details, please refer to the SDK RM.

4. Cache maintenance and data coherency

The cache brings a great performance boost, but the user must pay attention to the cache maintenance for data coherency.

4.1. Typical use case

To get better understanding on the cache maintenance and data coherency, this section describes a typical use case as an example: Playback an audio file stored in the external Flash. The subsystem inter-connection and program data flow is as below:

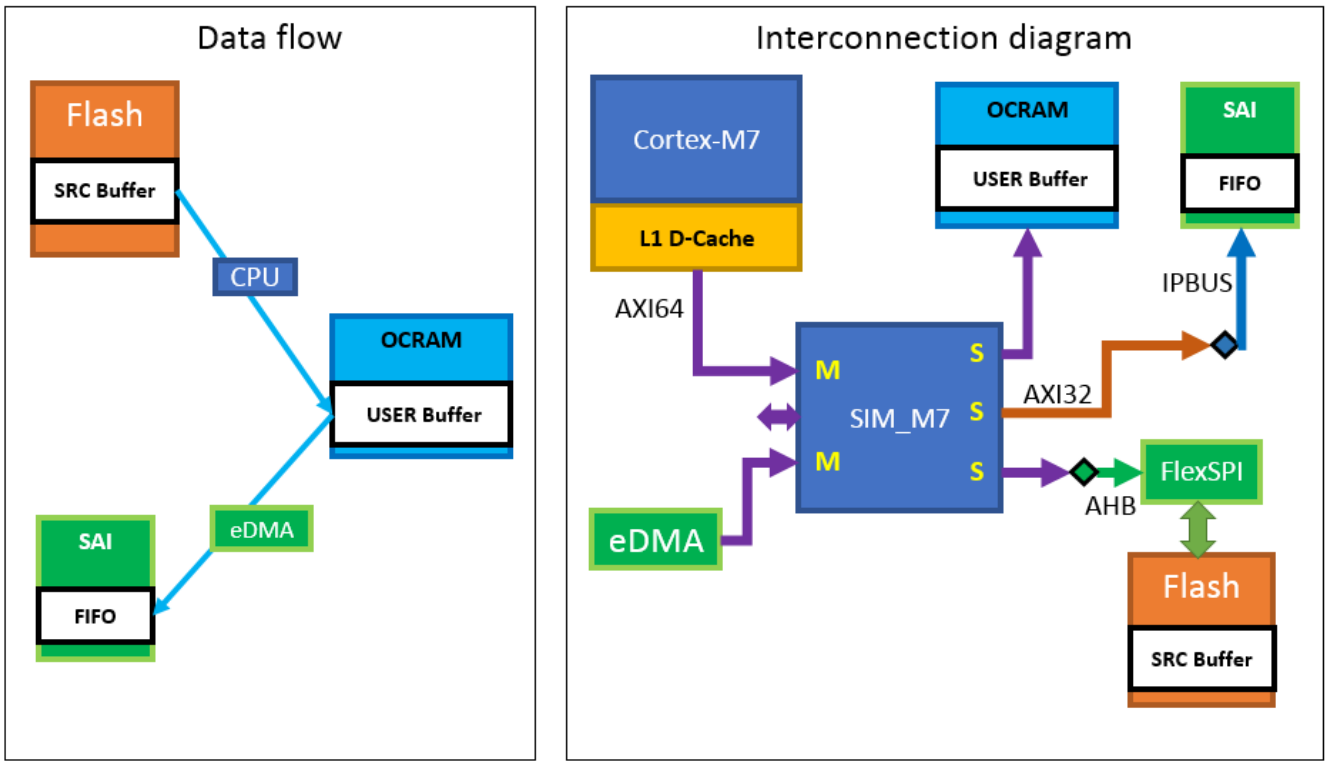

Figure 2. **Data flow & Subsystem diagram**

The CPU read the audio file content in SRC buffer through the L1 D-Cache, and decode the PCM frame data, write into OCRAM's USER buffer. After USER buffer is full, eDMA is started to copy the PCM frame data into the FIFO inside the SAI IP module. Then SAI shift out the FIFO data to SAI bus for audio playback. When CPU write the frame data to OCRAM with L1 cache enabled, the data may only be written to the cache as default cache policy for OCRAM is Write-Back. Then eDMA transfers the data to SAI FIFO is incorrect, and the data coherency problem occurs.

To avoid such data coherency issue, here're some solutions:

1. Perform a D-Cache clean operation after CPU writing data to OCRAM.
2. Configure the OCRAM memory region cache policy from Write-Back to Write-Through in MPU before this write started.
3. Configure the OCRAM memory region cache policy to non-cacheable in MPU.
4. Configure the OCRAM memory region as shareable in MPU, which means non-cacheable.

4.2. Cache maintenance in SDK Driver

The following drivers in the SDK maintain the data coherency of the cache:

1. **Ethernet**

In the ENET, a unified DMA (uDMA) engine is designed, it optimizes data transfer between the ENET core and the SoC, and supports an enhanced buffer descriptor programming model to support IEEE 1588 functionality.

2. **uSDHC**

   In the SD Host Controller Standard, a new DMA transfer algorithm called the ADMA (Advanced DMA) is designed.

   User can pass cacheable buffers to those drivers, drivers takes care of the data coherency. For other cases that uses DMA, user should take care of the data coherency by cache operations. Please refer to the next section.

4.3. Cache maintenance by application

There're two ways to do cache maintenance in application.

4.3.1. Use cacheable buffers

Normally buffers on the OCRAM, SDRAM are Cacheable and Write-Back. It can be in the stack, static section or allocated from heap. To use such buffer as DMA source, user must perform a DCACHE clean operation is done before DMA started, this makes sure all of the data are committed to the memory from cache. If buffer is used as DMA receive destination, a DCACHE invalidate operation must be done after DMA completed and before CPU or other masters read. The buffer address should be L1 cache line size aligned (32 bytes in i.MXRT).

4.3.2. Use non-cacheable buffers

Use non-cacheable buffer would make life easier, which can avoid the cache data coherency problem. But the side-effect is the performance of accessing the buffer is not good as cacheable ones if CPU access them multiple times.

To make buffers non-cacheable, user must configure at least one region of memory as non-cacheable attribute in MPU, and put the buffers into this region by the linker of toolchain.

Steps to do in application:

1. **Buffer definition**

   SDK provides the below two macros for application to define buffers (variable) in the "Noncacheable" section of the program:

```
AT_NONCACHEABLE_SECTION_ALIGN(var, alignbytes)
AT_NONCACHEABLE_SECTION(var)
```

The first macro is to define the buffer (var) with start address aligned by alignbytes. The second macro is to define the buffer (var) with start address aligned by compiler automatically, normally 4-

bytes aligned. Some use cases need application to explicit define the buffer aligned with special bytes. Like the framebuffer for eLCDIF, 8-bytes aligned is required, e.g.:

```
AT_NONCACHEABLE_SECTION_ALIGN(static uint8_t buffer[256], 8);
```

2. Linker file

Add the NCACHE_VAR block (NonCacheable section) with size for user buffers size requirement. Put this NCACHE_VAR block into the SDRAM region, e.g. very beginning of the SDRAM.

```
define symbol m_sdram_start          = 0x80000000;
define symbol m_sdram_end            = 0x80FFFFFF;

define region SDRAM_region = mem:[from m_sdram_start to m_sdram_end];

define block NCACHE_VAR with size = 2*1024*1024, alignment = 1024*1024   { section
NonCacheable };


place in SDRAM_region                  { first block NCACHE_VAR };
```

3. MPU configurations

Before the user can configure any regions, the MPU must be disable first by ARM_MPU_Disable() function. To configure the region, ARM_MPU_SetRegionEx() function take first parameter as Region Number, second parameter as base address of the memory want to configure. The third parameter is the Region Attributes and Size, the macro is defined as below:

ARM_MPU_RASR(XN, AP, TEX, S, C, B, SRD, Size)

- XN/AP/TEX/S/C/B parameters are exactly same as the attributes defined in section 2.2
- SRD means disable the Subregion, which used in the region overlap case. Here just ignore it and set to 0.
- Size is defined as: ***Region size in bytes = $2^{(SIZE+1)}$***

The base address of the region passed to ARM_MPU_SetRegionEx() must be aligned with the size in bytes. ARM_MPU_SetRegionEx() can be called multiple times to configure for different memory regions with unique Region Number. The MPU supports up to 16 regions. After memory regions been configured, the ARM_MPU_Enable() is called to enable MPU. The parameter of ARM_MPU_Enable() is set to 0x4, which enables use of the default memory map as background region.

For example, configure the SDRAM (start from 0x80000000) first 2MB region as non-cacheable:

```
ARM_MPU_Disable();

// Configure for a non-cache region in beginning of SDRAM (2MB)
ARM_MPU_SetRegionEx(0, 0x80000000,
                ARM_MPU_RASR(1, ARM_MPU_AP_FULL, 0x1, 0, 0, 0, 0,
                                ARM_MPU_REGION_SIZE_2MB));

ARM_MPU_Enable(0x4);
```

5. Constraint Speculative Prefetch

As Cortex-M7 support speculative prefetch feature, which can do speculative accesses to memory locations with Normal Memory attribute at any time, and if prefetching happens on invalid address, it will generate bus fault, so it must to avoid this issue occur.

Since speculative fetch will never be performed to Strongly Ordered or Device Memory (refer to *2.3* for memory type), so it is way to configure the MPU to constraint its behavior.

Need to consider MPU configuration as below:

- Configure the used memory as Normal Memory
  I.MXRT reserves memory address for some device, need configure the used address space as Normal Memory. such as, for SDRAM, need to configure the valid address space with Normal Memory, and MPU configuration as below:

```
/* Region 7 setting */
ARM_MPU_SetRegion ( ARM_MPU_RBAR ( 7 , 0x80000000U ) ,
                    ARM_MPU_RASR(0, ARM_MPU_AP_FULL, 0, 0, 1, 1, 0,
                   ARM_MPU_REGION_SIZE_32MB) ) ;
```

- Configure all unused address space as Device Memory.
  As not all memory is used to one application, it requires to configure all unused memory to Device Memory, for example, if it don't install external flash (by FlexSPI interface) on board, need to configure corresponding address region to Device Memory type as below.

```
/* Region 2 setting */
ARM_MPU_SetRegion ( ARM_MPU_RBAR ( 2 , 0x60000000U ) ,
                    ARM_MPU_RASR(0, ARM_MPU_AP_FULL, 2, 0, 0, 0, 0,
                   ARM_MPU_REGION_SIZE_512MB) ) ;
```

- Need to ensure all memory space get the correct configuration on MPU.
  Please configure correct memory attribute according to application, check the used/unused memory and valid address space, assign the correct memory type to avoid issue caused by

speculative prefetch, also I.MXRT SDK provide the MPU initialization driver(BOARD_ConfigMPU(void)), which is an example to configure MPU.

6. Conclusion

In summary, to use i.MXRT L1 Cache in a correct and efficient way, there are several recommendations and tips:

- Put critical code and data into TCM, like vector table. Which is the fastest way for CPU to access the code and data.
- Always call the CMSIS cache function or SDK cache driver API to cache operations. This make sure cache is cleaned before disabled, and invalided before enabled, to avoid unpredictable issue.
- If the software is using cacheable memory regions for the DMA source/or destination buffers. The software must trigger a cache clean before starting a DMA operation to ensure that all the data are committed to the subsystem memory. After the DMA transfer complete, when reading the data from the peripheral, the software must perform a cache invalidate before reading the DMA updated memory region.
- Always recommended to use non-cacheable regions for DMA buffers. The software can use the MPU to configure a non-cacheable memory region to use as a shared buffer between the CPU and DMA. For example:
    - The frame buffer for eLCDIF display
    - The input and output buffer for PXP channel
- When using FlexSPI for external NOR Flash read by AHB bus, cacheable memory would cause problem. Because the Flash erase and program operation is go through the IP command, but not AHB bus. A cache invalid operation is needed before CPU read the FlexSPI memory map after any erase and program completed.

7. Reference

- ARM Cortex-M7 Processor User Guide (Revision: r1p1)
- ARM Cortex-M7 Processor Technical Reference Manual (Revision: r1p1)
- i.MX RT1050 Processor Reference Manual
- Kinetis SDK v.2.2 API Reference Manual (In the SDK release package)
- Cache (Computer) - https://en.wikipedia.org/wiki/Cache_(computing)

8. Revision history

Table 4. **Revision history**

Revision number	Date	Substantive changes
0	08/2017	Initial release
1	12/2017	Add chapter 5 and 2.5

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

.

Document Number: AN12042
Rev. 1
12/2017