Application note

Level 1 cache on STM32F7 Series and STM32H7 Series

Introduction

The memory protection unit (MPU) in the Cortex®-M7 processor allows the modification of the Level 1 (L1) cache attributes by region. The cache control is done globally by the cache control register, but the MPU can specify the cache mode and whether the access to the region can be cached or not.

In some cases, the cached systems need to ensure data coherency between the core and the main memory when dealing with shared data.

This application note describes the level 1 cache behavior and gives an example showing how to ensure data coherency in the STM32F7 Series and STM32H7 Series when using the L1-cache.

For more details about the MPU and how to set the memory attributes according to the memory type and the cache policy, the user can refer to the following documents available on http://www.st.com:

- STM32F7 Series and STM32H7 Series Cortex®-M7 processor programming manual (PM0253).
- Managing memory protection unit (MPU) in STM32 MCUs (AN4838).

1 General information

This document applies to Arm®-based devices.

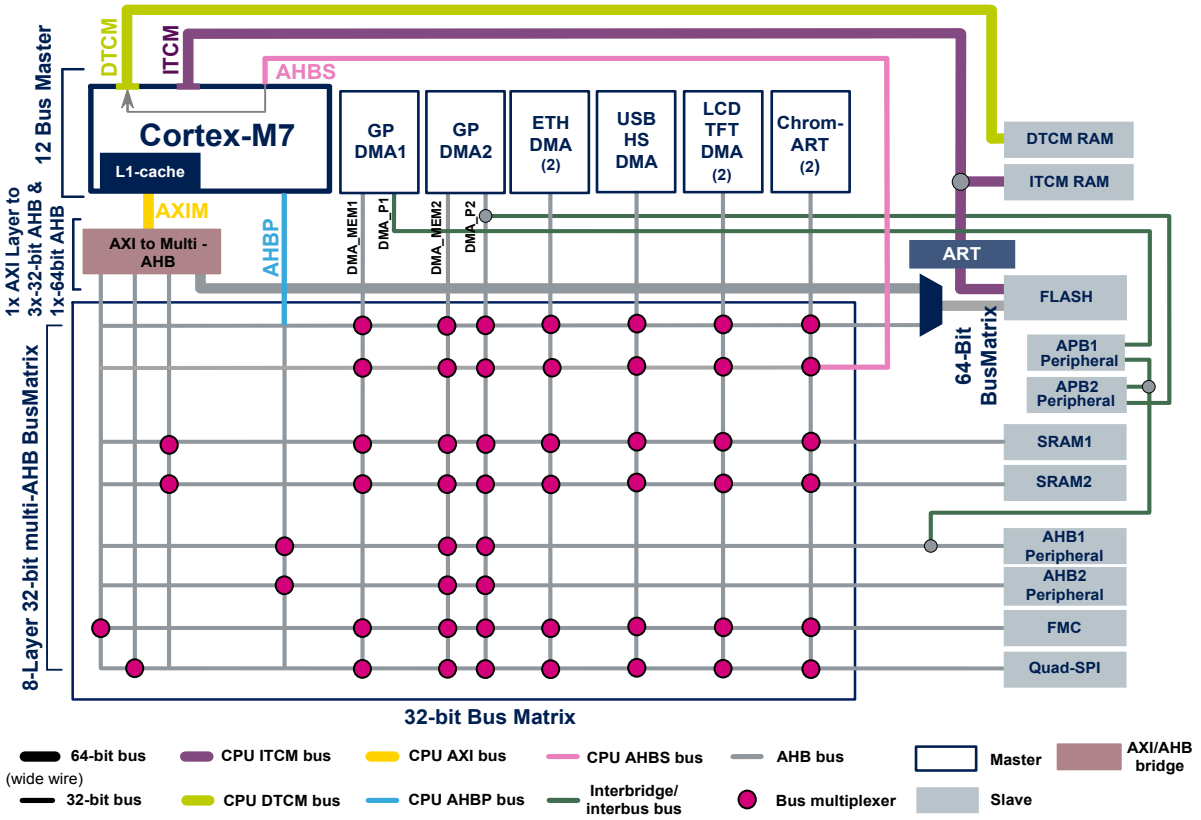
arm

2 Cache control

The STM32F7 Series and STM32H7 Series devices include up to 16 Kbytes of L1-cache both for the instructions and the data. An L1-cache stores a set of data or instructions near the CPU, so the CPU does not have to keep fetching the same data that is repeatedly used, such as a small loop.

Figure 1. STM32F7 Series system architecture illustrates an example of the system architecture in the STM32F7 Series.

Figure 1. STM32F7 Series system architecture



Since the memory accesses to the subsystem can take multiple cycles (especially on the external memory interfaces with multiple wait states), the caches are intended to speed up the read/write operation to the memory. The idea is that both operations can be optimized if the data are available locally (in an area that only takes one cycle to access). The bus accesses to the subsystem memory, that take more than one CPU cycle to execute, are differed from the CPU pipeline instruction stream execution. This allows a big performance boost. A cache is normally implemented using sets of lines where a line is just a short segment of memory. The number of lines in a set is called x-way associative. This property is set in the hardware design.

A read pulls the location from the memory only the first time that the location is accessed. A write either pushes the value through the memory (write-through mode) or just places it in the cache for a later write (write-back mode). Each mode has pros and cons in regard to the performance that must be weighed in accordance with the application.

If the memory is write-back, the cache line is marked as dirty, and the write is only performed on the AXIM interface when the line is evicted. When a dirty cache line is evicted, the data are passed to the write buffer in the AXIM interface to be written to the external memory system.

The L1-caches on all Cortex®-M7s are divided into lines of 32 bytes. Each line is tagged with an address. The data cache is 4-way set associative (four lines per set) and the instruction cache is 2-way set associative. This is a hardware compromise to keep from having to tag each line with an address.

A cache hit is when an address falls anywhere within a given set of lines. So the hardware has to do fewer address compares to find out if that address is cached. If there is a hit, the cache value is used for a read, or the value is stored for a write. If there is a miss, a new line is allocated and tagged, and the cache is filled in from either read or write accesses. If all the lines are allocated, the cache controller runs the line eviction process, where a line is selected (depending on replacement algorithm) cleaned/invalidated, and reallocated. The data cache and Instruction cache implement a pseudo-random replacement algorithm.

The L1-cache can be a performance booster when used in conjunction with memory interfaces on AXI bus. This must not be confused with memories on the Tightly Couple Memory (TCM) interface, which are not cacheable. Any normal memory area can be cacheable, as described above, but the biggest gains are seen on memories accessed by the AXI bus such as the internal Flash memory, internal SRAMs and external memories attached to the FMC or Quad-SPI controllers.

There are four basic cache operations: enable, disable, clean, and invalidate. Dedicated APIs are available in the STM32F7 and STM32H7 Cube firmware packages for these operations, reducing the development time.

2.1 Accessing the Cortex®-M7 cache maintenance operations using CMSIS

The CMSIS cache functions defined in core_cm7.h are illustrated in Table 1. CMSIS cache functions.

Table 1. CMSIS cache functions

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

- **Cache clean**: the operation writes back dirty cache lines to the memory (an operation sometimes called a flush).
- **Invalidate cache:** the operation marks the contents as invalid (basically, a delete operation).

3 Cache operation

Using the cache is simple at the most basic level. The user needs just to set up the region in the MPU and enable the cache via the CMSIS function listed above. For example the user can set up things using either write-back or write-through.

- **Write-back:** the cache does not write the cache contents to the memory until a clean operation is done.
- **Write-through:** triggers a write to the memory as soon as the contents on the cache line are written to. This is safer for the data coherency, but it requires more bus accesses. In practice, the write to the memory is done in the background and has a little effect unless the same cache set is being accessed repeatedly and very quickly. It is always a tradeoff.

3.1 STM32F7 and STM32H7 default settings

By default, the MPU is disabled. In this case, the cache setting is defined as a default address map.

Table 2. Memory region shareability and cache policies

Address range	Memory region	Memory type	Shareability	Cache policy
0x00000000-0x1FFFFFFF	Code	Normal	Non-shareable	WT
0x20000000-0x3FFFFFFF	SRAM	Normal	Non-shareable	WBWA
0x40000000-0x5FFFFFFF	Peripheral	Device	Non-shareable	-
0x60000000-0x7FFFFFFF	External RAM	Normal	Non-shareable	WBWA
0x80000000-0x9FFFFFFF				WT
0xA0000000-0xBFFFFFFF	External Device	Device	Shareable	-
0xC0000000-0xDFFFFFFF			Non-shareable	
0xE0000000-0xE00FFFFF	Private peripheral bus	Strongly-ordered	Non-shareable	-
0xE0100000-0xFFFFFFFF	Vendor-specific system	Device	Non-shareable	-

3.2 Example for cache maintenance and data coherency

The purpose of this example is to get familiarized with the ARM® Cortex®-M7 data cache coherency.

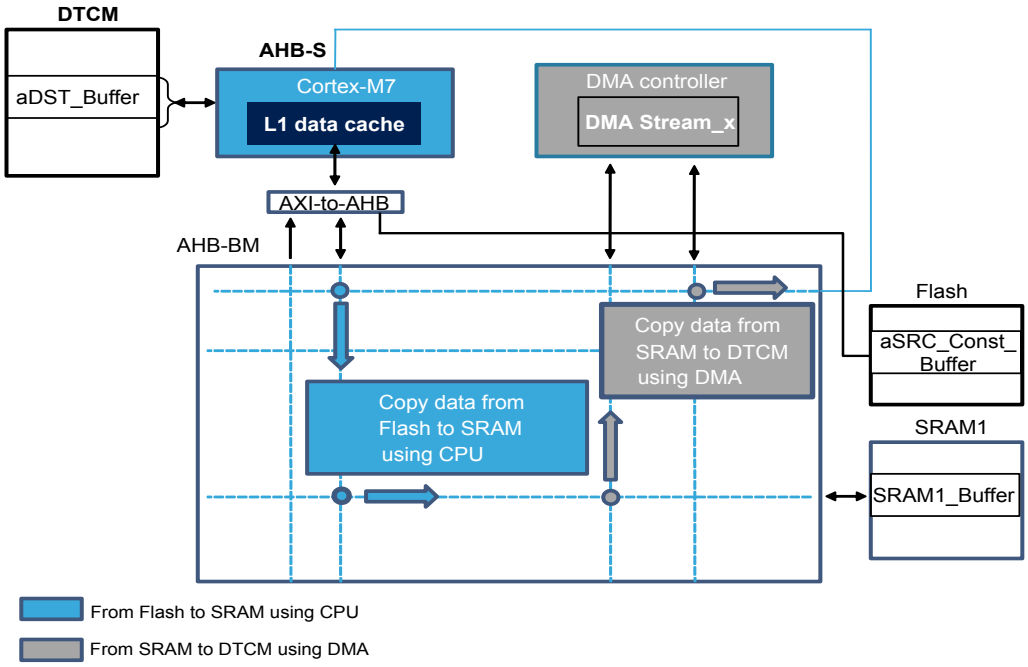
At first the CPU copies 128-byte constant pattern from the Flash memory “aSRC_Const_Buffer” to the SRAM1 temporary buffer “pBuffer”

Then the CPU configures and enables the DMA to perform a memory-to-memory transfer to copy from the SRAM1 “pBuffer” to the destination buffer “aDST_Buffer” defined in DTCM RAM.

Finally the CPU compares the data read by DMA aDST_Buffer with a constant pattern from the Flash memory (aSRC_Const_Buffer).

Figure 2. Data transfer paths illustrates the data transfer paths.

Figure 2. Data transfer paths



The purpose is to show the impact on data coherency between the CPU and DMA when accessing a cacheable memory region with the write-back attribute set.

By default after reset, the data and instruction cache are disabled. When the data cache is disabled the data transfer between SRAM1 and DTCM RAM is done successfully in the described scenario above.

When enabling the data cache before running the described transfer scenario, a data mismatch is detected between the data in “aDST_Buffer” (DMA destination buffer in DTCM) and “aSRC_Const_Buffer” (CPU data source buffer in the Flash memory).

When using the L1-cache there is always an ongoing problem, sometimes called cache coherency. This matter crops up when multiple masters (CPU, DMAs...) share the memory. If the CPU writes something to an area that has a write-back cache attribute (example SRAM1), the write result is not seen on the SRAM as the access is buffered, and then if the DMA reads the same memory area to perform a data transfer, the values read do not match the intended data.

- **Solution 1:** to perform a cache maintenance operation after writing data to a cacheable memory region, by forcing a D-cache clean operation by software through CMSIS function SCB_CleanDCache() (all the dirty lines are write-back to SRAM1).
- **Solution 2:** in order to ensure the cache coherency, the user must modify the MPU attribute of the SRAM1 from write-back (default) to write-through policy.
- **Solution 3:** to modify the MPU attribute of the SRAM1 by using a shared attribute. This prevents by default the SRAM1 from being cached in D-cache.
- **Solution 4:** to perform a cache maintenance operation, by forcing write-through policy for all the writes. This can be enabled by setting force write-through in the D-Cache bit in the CACR control register.

The data coherency between the core and the DMA is ensured by:

1. Either making the SRAM1 buffers not cacheable
2. Or making the SRAM1 buffers cache enabled with write-back policy, with the coherency ensured by software (clean or invalidate D-Cache)
3. Or modifying the SRAM1 region in the MPU attribute to a shared region.
4. Or making the SRAM1 buffer cache enabled with write-through policy.

Another case is when the DMA is writing to the SRAM1 and the CPU is going to read data from the SRAM1. To ensure the data coherency between the cache and the SRAM1, the software must perform a cache invalidate before reading the updated data from the SRAM1.

For more details about the cache maintenance operations the user can refer to cache maintenance operations section in STM32F7 Series and STM32H7 Series Cortex®-M7 processor programming manual (PM0253).

4 Mistakes to avoid and tips

- After reset, the user must invalidate each cache before enabling it, otherwise an UNPREDICTIBLE behavior can occur.
- When disabling the data cache, the user must clean the entire cache to ensure that any dirty data is flushed to the external memory.
- Before enabling the data cache, the user must invalidate the entire data cache if the external memory might have changed since the cache was disabled.
- Before enabling the instruction cache, the user must invalidate the entire instruction cache if the external memory might have changed since the cache was disabled.
- 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 better to use non-cacheable regions for DMA buffers. The software can use the MPU to set up a non-cacheable memory block to use as a shared memory between the CPU and DMA.
- Do not enable cache for the memory that is being used extensively for a DMA operation.
- When using the ART accelerator, the CPU can read an instruction in just 1 clock from the internal Flash memory (like 0-wait state). So I-cache cannot be used for the internal Flash memory.
- When using NOR Flash, the write-back causes problems because the erase and write commands are not sent to this external Flash memory.
- If the connected device is a normal memory, a D-cache read is useful. However, If the external device is an ASIC and/or a FIFO, the user must disable the D-cache for reading.

Revision history

Table 3. **Document revision history**

Date	Revision	Changes
23-Mar-2016	1	Initial release.
06-Mar-2018	2	Added STM32H7 Series in the whole document. Updated Figure 1. STM32F7 Series system architecture. Updated Figure 2. Data transfer paths. Added Section 1 General information.

Contents

List of tables

List of figures