APPLICATION NOTE

How to Optimize Usage of SAM S70/E70/V7x Architecture

Atmel | SMART SAM S70/E70

Introduction

The purpose of this application note is to understand the architecture Cortex®-M7 processor and the Atmel® | SMART SAM S70/E70/V7x devices and how to tune an application code to benefit from it and maximize performance.

Firstly a short introduction on the Cortex-M7 core will be made with details on the implementation done in SAM S70/E70/V7x devices. Then the document will focus on the architecture of the SAM S70/E70/V7x itself. The last part of the application note will explain how to enable and properly use the features previously introduced. A concrete example will be used to illustrate it: an application code performing FFT computation will be successively executed out of different memories to have a better understanding on the impact on performance.

Table of Contents

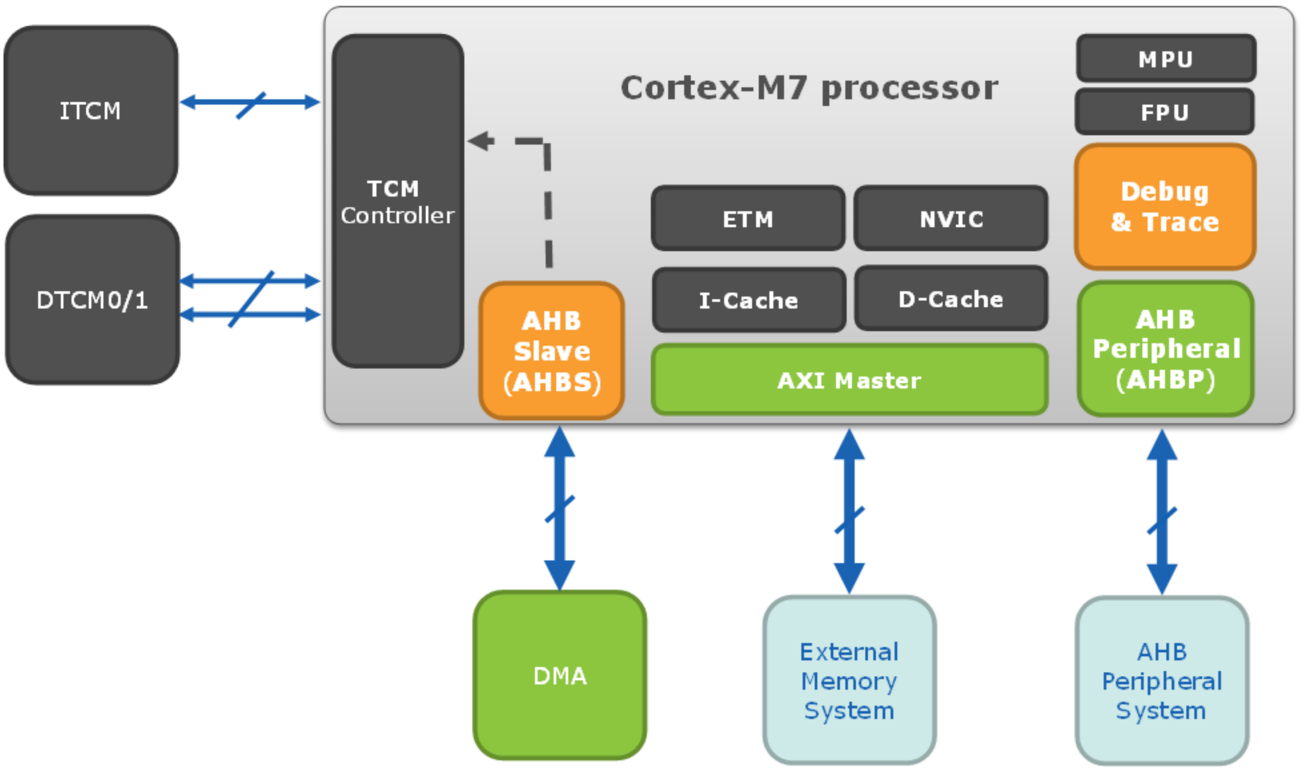
1 Introduction to ARM Cortex-M7 Processor

The ARM® Cortex-M7 processor is the high-end processor of the Cortex-M family, providing 5 CoreMark®/MHz and up to 630 DMIPS. It delivers the best performance level thanks to various new features:

- Six-stage superscalar pipeline allowing parallel execution of instructions (dual instruction issue)
- Instruction and Data Tightly Coupled Memories (ITCM and DTCM respectively) with access at processor clock speed with no wait state penalty and deterministic behavior
- Inner Data and Instruction caches to compensate wait state penalty when executing code out of external memory
- DSP extension and single/double precision Floating Point Unit (FPv5) with extended instruction set
- Embedded Trace Module (ETM) with instruction and data trace capability
- Various memory interfaces to increase overall bandwidth:
  - 64-bit AXI Master Interface to access memories and peripherals. This memory interface has been optimized for throughput
  - 32-bit AHB Peripheral interface (AHBP) to access low-latency peripherals rather than memory (peripheral data access only)
  - 32-bit AHB Slave interface (AHBS) providing a direct access path between the DMA and TCM
  - 32-bit AHB Debug port providing a debug interface (JTAG or SWD interfaces)

The Cortex-M7 processor is based on the ARMv7-M architecture so it is binary compatible with other Cortex-M processors.

Figure 1-1. Cortex-M7 Processor Overview



1.1 Six-stage superscalar pipeline

The Cortex-M7 has a six-stage, in order, dual-issue superscalar pipeline with branch prediction.

Thanks to the dual instruction issue, the Cortex-M7 processor is able to execute two instructions in parallel. The pipeline features an optional float pipeline so that floating point instructions can be dual issued with integer instructions as well.

In addition, memory accesses are interleaved with computation to reduce latency, which is a major improvement compared to previous Cortex-M processors neither dual-issue nor interleaving in Cortex-M0/M3/M4 processors).

Integer MAC instruction execution has been also improved and takes one cycle only (from two to four cycles for floating MAC instructions) so it is used whenever possible.

As a consequence Cortex-M7 processor doubles the performance of the Cortex-M4 processor when executing math functions (FFT, FIR etc.).

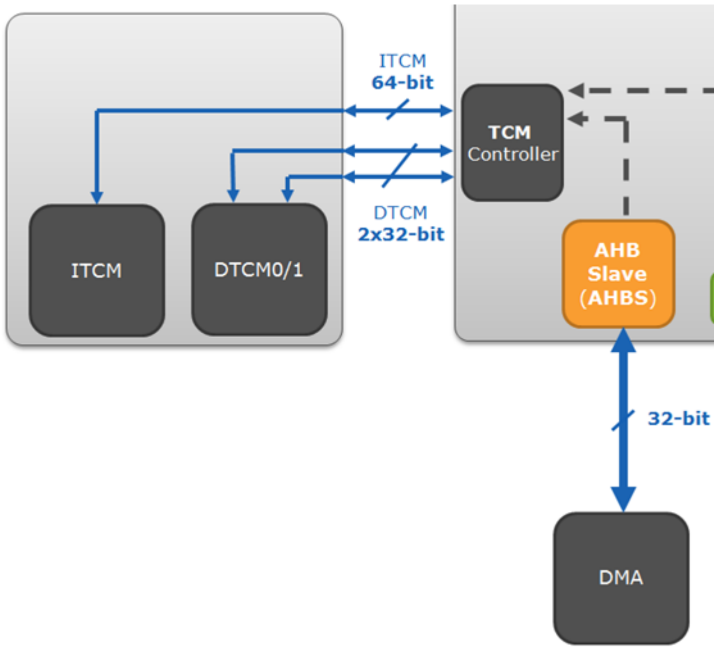
1.2 Instruction and Data Tightly Coupled Memories

The TCM controller provides a direct access between the processor and two memory areas: Instruction TCM (ITCM) and Data TCM (DTCM). The size of these memory areas can be up to 16Mbytes each, but depends on the silicon vendor integration (up to 128KB each on the SAM S70/E70/V7x devices - see later for more information).

As shown in Figure 1-2, the interface between the processor and the ITCM is a 64-bit interface, so that the processor can fetch two 32-bit instructions in a single access, and thus benefit from the dual-issue capability of the pipeline.

The DTCM interface is a dual 32-bit interleaved interface (DTCM0 and DTCM1), thus concurrent accesses (from the DMA and the core for instance) can be optimized.

Figure 1-2. Tightly Coupled Memories



Instructions and data located in TCM can be directly accessed at processor speed (e.g. up to 300MHz on the SAM S70/E70) with no wait state penalty, as opposed to the other memories such as the flash, which are accessed at bus speed through the AXI master interface.

The purpose of the TCM memories is to store critical part of code which needs to be processed as fast as possible. In addition TCM has a deterministic behavior, which makes it perfectly suitable for RTOS-based applications.

1.3 Instruction and Data Cache

Cortex-M7 embeds an Instruction (I-cache) and Data cache (D-cache) to compensate wait state penalty when executing code out of external memory (typically flash):

- Instruction cache is 2-way associative, up to 64kB with optional ECC (16kB in SAM S70/E70/V7x)
- Data cache is 4-way associative, up to 64kB with optional ECC (16kB in SAM S70/E70/V7x)

I-cache and D-cache are disabled by default and must be enabled by default in the application code.

1.4 Floating Point Unit

The Cortex-M7 processor can optionally integrate a Floating Point Unit (FPU). This FPU, corresponding to the FPv5 extension, share the same instruction set as the previous version (FPv4) which was implemented on Cortex-M4 processor and adds the double–precision operand support. It also introduces new instructions such as rounding functions.

For more information on floating point instructions, you can refer to the Cortex-M7 Devices Generic User Guide from ARM: http://infocenter.arm.com/help/topic/com.arm.doc.dui0646a/CHDHHAJF.html.

1.5 Memory Interface

As shown in Figure 1-1, Cortex-M7 core has four main memory interfaces.

1.5.1 AXI Master Interface

As said in introduction the Cortex-M7 processor features a new 64-bit interface running at processor frequency. This interface is for both instruction and data accesses and dedicated for on-chip and off-chip memories and devices (typically flash or RAM).

This interface has been optimized for performance: AXI accesses are not made in a predictable order as this bus can re-order instruction or data to reduce latency and increase bandwidth.

If predictability is critical when executing some part of an application code, it is recommended to:

- Configure the Memory Protection Unit (MPU) to define memory regions with proper attributes. For more information on MPU configuration one can refer to the following application note:
  http://www.atmel.com/Images/Atmel-42128-AT02346-Using-the-MPU-on-Atmel-Cortex-M3-M4-based-Microcontroller_Application-Note.pdf
  A typical MPU configuration example is also included in the software examples from the SAM S70/E70/V7x Software Package.
- Use memory barriers. More information can be found in a dedicated application note from ARM:
  http://infocenter.arm.com/help/topic/com.arm.doc.dai0321a/DAI0321A_programming_guide_memory_barriers_for_m_profile.pdf

1.5.2 AHB Peripheral Interface

The AHB Peripheral interface (AHBP) is dedicated to access low-latency peripherals rather than memory. Note that AHBP does not support instruction fetch (which is performed through the AXI master interface), but only data transfers. This interface has been added to avoid overloading the AXI master interface with additional data transfers and so to increase the overall bandwidth.

Unlike AXI master interface, there is no optimization done and speculative access is not supported (e.g. buffering is ordered). In addition bursts are not supported so only single access can be performed. For burst access to peripheral it is recommended to use the DMA to do so.

1.5.3 AHB Slave Interface

The AHB Slave port (AHBS), also called DMA slave port, provides a system access between the DMA and the TCM through a 32-bit interface. This interface is also usable when processor is in sleep state.

Thanks to this interface both processor and DMA can access the TCM in parallel, thus bandwidth is maximized when running high resources demanding applications.

1.5.4 AHB Debug Interface

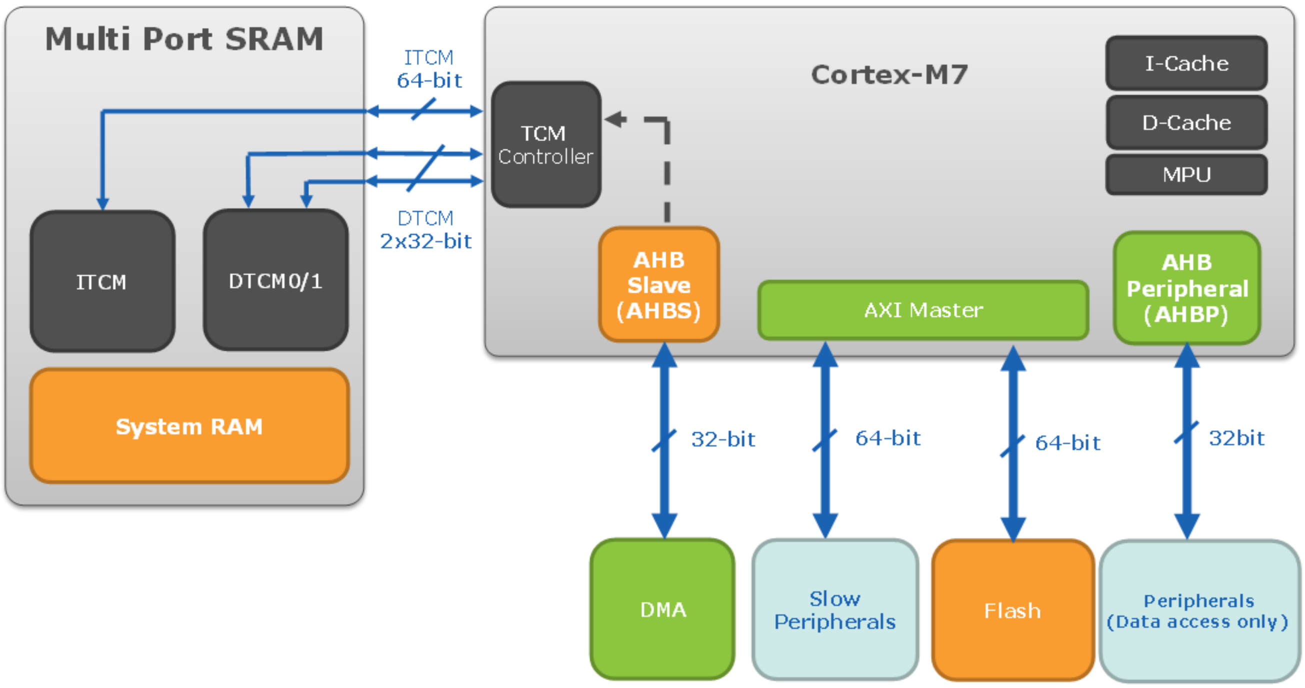
The Cortex-M7 processor implements a complete hardware debug solution, accessible through the AHB Debug interface (AHBD). This provides high system visibility of the processor and memory through either a traditional JTAG port or a 2-pin Serial Wire Debug (SWD) port.

2 SAM S70/E70/V7x Devices Architecture and Highlights

2.1 High Performance Implementation

In order to support larger flash memory range and benefit from the AXI bus performance, Atmel chose to implement the Cortex-M7 processor according to the *High Performance* configuration recommended by ARM. Figure 2-1 shows a simplified diagram of this implementation, with the example of the SAM S70/E70/V7x devices.

Figure 2-1. Cortex-M7 Processor Implementation



Unlike the *Simple Microcontroller* implementation application code can run from flash (connected to the AXI master interface) or TCM, and thus benefit of large data storage (flash is up to 2MB) and performance boost with the TCM running at 300MHz. This architecture also enables the use of I/D cache to accelerate code execution when running out of flash.

The table below shows the options selected by Atmel when implementing the Cortex-M7 processor in the SAM S70/E70/V7x devices:

Features	Configurable options	SAM S70/E70/V7x implementation
FPU	No FPU Single Precision (SP) only SP and DP	Single and Double Precision FPU
ITCM max size	No ITCM / 4kB-16MB	128kB
DTCM max size	No ITCM / 4kB-16MB	128kB
I-cache size	4, 8, 16, 32, 64kB	16kB
D-cache size	4, 8, 16, 32, 64kB	16kB
AHB Peripheral size (AHBP)	64, 128, 256, 512MB	512MB
ECC support on caches	Implemented or not	Implemented

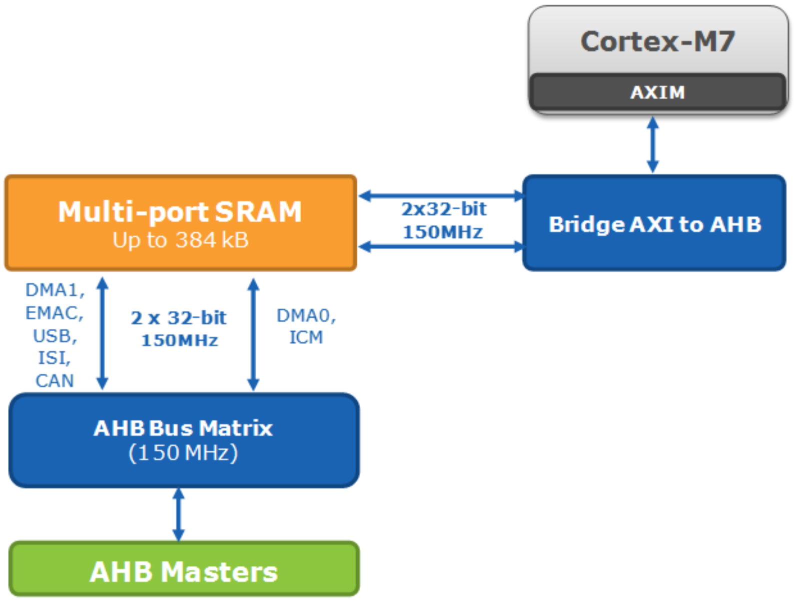
Features	Configurable options	SAM S70/E70/V7x implementation
MPU	0, 8, 16 regions	16 regions
Interrupts	1-240	72
Debug Watchpoints and Breakpoints	2 data watchpoints + 4 breakpoints 4 data watchpoints + 8 breakpoints	4 data watchpoints + 8 breakpoints
ITM and DWT Trace	Implemented or not	Implemented
ETM	No ETM ETM instruction only ETM instruction and data	ETM Instruction Trace only
Cross Trigger Interface (CTI) Wake-up Interrupt Controller (WIC)	Implemented or not	Not Implemented

In the next coming paragraphs a review of the key features will be done to understand how SAM S70/E70/V7x devices have been designed to optimize performance.

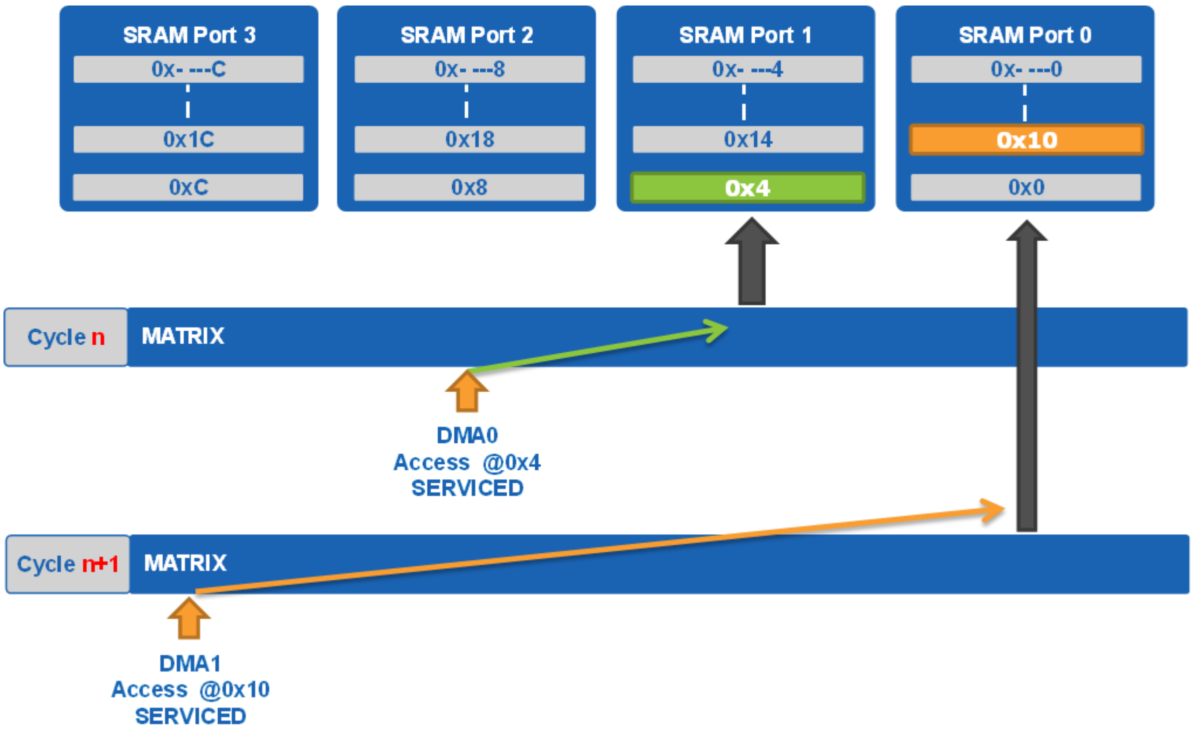
2.2 Multi-port SRAM

The SAM S70/E70/V7x devices feature a multi-port SRAM, which can be up to 384kB. This SRAM spaces operates at bus clock (i.e. processor clock/2 = up to 150MHz) and has four ports to optimize the bandwidth and latency. As shown in Figure 2-2, two ports are dedicated to the Cortex-M7 processor and two ports are shared by AHB masters (Central DMA, EMAC DMA, USB DMA etc.).

Figure 2-2. Multi-port SRAM Implementation



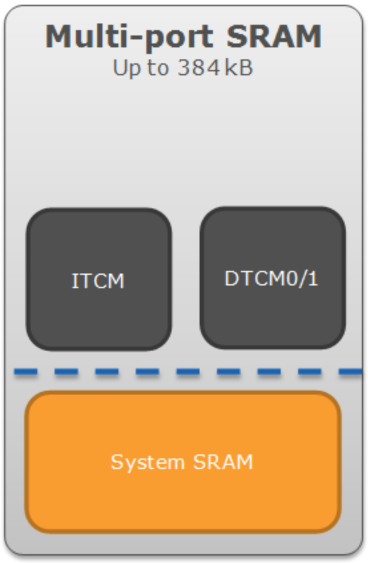
The purpose of the multi-port capability is to decrease the latency when several masters try to access the SRAM simultaneously: the integrated controller manages interleaved addressing of SRAM blocks so that another master will be able to access it on the next cycle. As an example, when a 16-word burst is performed by the DMA, another master will be able to access the SRAM on the next cycle (n+1) and not on the (n+16) one.



The multi-port SRAM is split in two regions:

- System RAM region
- TCM region

Figure 2-3. Multi-port SRAM



The size of the TCM region can be configured through NVM bits, and the remaining SRAM size is automatically assigned to System RAM region.

2.2.2 System RAM

System RAM is accessed by AHB masters (peripherals, DMA) through the AHB matrix or the Cortex-M7 core through a privileged access (see Figure 2-2) at bus clock frequency (up to 150MHz).

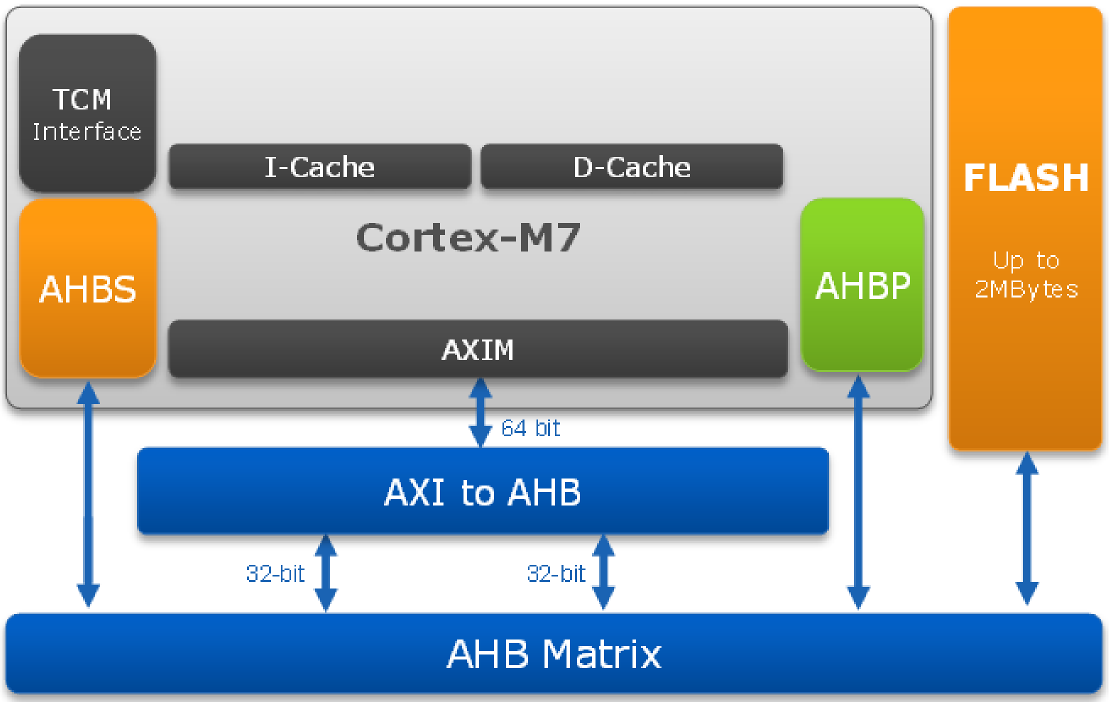
2.2.3 Tightly Coupled Memories

As explained previously ITCM and DTCM can be directly accessed by the processor at 300MHz. Transfers between DMA and TCM is done though the dedicated AHBS interface to maximize bandwidth and free CPU resources.

2.3 Internal Flash

Internal flash is accessed through the AXI Master and AHB matrix at bus clock. Wait states must be added depending on the operating frequency (up to five wait states when processor is running at maximum frequency), but thanks to I-cache and D-cache penalty can be compensated.

Figure 2-4. Connection Between Flash and Memory Interface



3 How to Benefit from the SAM S70/E70/V7x Architecture to Optimize Performance

This chapter gives more details on how to properly enable and configure the features previously introduced to reach the maximum performance level. Usage of the Floating Point Unit, I- and D-cache and Tightly Coupled Memories will be discussed in this part, illustrated with code snippets for Atmel Studio, Keil and IAR™ users. More information will be also given on how to optimize the load/store process and boost performances.

3.1 Use of Floating Point Unit

By default Floating Point Unit is not enabled and developers have to enable it in the application code and make sure FPU instructions will be used by the compiler.

3.1.1 Enable the Floating Point Unit

When developing an application code requiring the use of the FPU, one have to ensure FPU is enabled by software using the following code sequence:

```
/** Address for ARM CPACR */
#define ADDR_CPACR 0xE000ED88

/** CPACR Register */
#define REG_CPACR  (*((volatile uint32_t *)ADDR_CPACR))

/**
* \brief Enable FPU
*/
__always_inline static void fpu_enable(void)
{
       irqflags_t flags;
       flags = cpu_irq_save();
       REG_CPACR |=  (0xFu << 20);
       __DSB();
       __ISB();
       cpu_irq_restore(flags);
}
```

Note: __DSB() is a memory barrier used to ensure that all data memory transfers are complete.
__ISB() is also a memory barrier used to flush the instruction pipeline.
More information can be found in Section 1.5.1 of this document.

3.1.2 Compiler Configuration

User has to make sure FPU instructions will be used when compiling the application code and configure the compiler accordingly. For ARM GCC compiler the following flags need to be added:

- `-mfloat-abi=hard` or `softfp`

  `softfp`: to be used only if the user intends to use same binary on Cortex-M4F and Cortex-M7F based product

  `hard`: in any other case

- `-mfpu=fpv5-sp-d16` or `fpv5-d16`

  `fpv5-sp-d16` is for single precision

  `fpv5-d16` is for double precision

In Atmel Studio, those flags are added in *Project Properties* menu > *Toolchain* tab > *ARM/GNU C Compiler* menu > *Miscellaneous*

For IAR users, there is no need to add flags and users simply have to select *VFPv5 single-precision* or *VFPv5 double-precision* in the project options (in *General Options* > *Target* tab > *FPU* drop-down menu).

3.2 Enable I-cache and D-cache

By default I-cache and D-cache are disabled and must be active in the application code, like the Floating Point Unit.

Below is a code example to enable it:

```
__STATIC_INLINE void SCB_EnableICache(void)
{
  #if (__ICACHE_PRESENT == 1)
    __DSB();
    __ISB();
    SCB->ICIALLU = 0;                // invalidate I-Cache
    SCB->CCR |=  SCB_CCR_IC_Msk;    // enable I-Cache
    __DSB();
    __ISB();
  #endif
}

_STATIC_INLINE void SCB_EnableDCache(void)
{
  #if (__DCACHE_PRESENT == 1)
    uint32_t ccsidr, sshift, wshift, sw;
    uint32_t sets, ways;

    ccsidr  = SCB->CCSIDR;
    sets    = CCSIDR_SETS(ccsidr);
    sshift  = CCSIDR_LSSHIFT(ccsidr) + 4;
    ways    = CCSIDR_WAYS(ccsidr);
    wshift  = __CLZ(ways) & 0x1f;

    __DSB();

    do {                        // invalidate D-Cache
         int32_t tmpways = ways;
         do {
              sw = ((tmpways << wshift) | (sets << sshift));
              SCB->DCISW = sw;
            } while(tmpways--);
        } while(sets--);
    __DSB();

    SCB->CCR |=  SCB_CCR_DC_Msk;    // enable D-Cache

    __DSB();
    __ISB();
  #endif
}
```

The two above functions are included in the software library provided for free on Atmel.com (SAM V71 / V70 / E70 / S70 Software Package).

3.3 Relocate Critical Part of Code in Tightly Coupled Memory

As explained previously Tightly Coupled Memories are intended to store critical parts of code which needs to be executed with a maximum performance. When determining which functions or data will be executed/stored out of TCM, it is important to be understand that power consumption increases according to its usage: the more TCM is used, the more power consumption increase. So user has to find the best trade-off between performance and power consumption and make sure his application does not exceed the power budget when moving code to TCM.

3.3.1 TCM Configuration

The first step is to configure the TCM size and enable it in the application code. This step is performed at startup before initializing the clocks and other peripherals.

TCM size is set with GPNVM bits 7 and 8 according to the below table.

GPNVM8	GPNVM7	ITCM/DTCM size [kB]	System RAM (348kB – TCMs size)
0	0	0 / 0	384
0	1	32 / 32	320
1	0	64 / 64	256
1	1	128 / 128	128

GPNVM bits are configured with specific EFC commands, as shown in the below code example:

```
/*Configure TCM sizes to: 128 kB ITCM - 128 kB DTCM (set GPNVM7 and GPNVM8)*/
EFC->EEFC_FCR = (EEFC_FCR_FKEY_PASSWD | EEFC_FCR_FCMD_SGPB | EEFC_FCR_FARG(7));
EFC->EEFC_FCR = (EEFC_FCR_FKEY_PASSWD | EEFC_FCR_FCMD_SGPB | EEFC_FCR_FARG(8));
```

Then TCM must be enabled by writing in the SCB register (System Control Block register) of the Cortex-M7 processor:

```
__STATIC_INLINE void TCM_Enable(void)
{
  __DSB();
  __ISB();
  SCB->ITCMCR = (SCB_ITCMCR_EN_Msk | SCB_ITCMCR_RMW_Msk | SCB_ITCMCR_RE-
TEN_Msk);
  SCB->DTCMCR = (SCB_DTCMCR_EN_Msk | SCB_DTCMCR_RMW_Msk | SCB_DTCMCR_RE-
TEN_Msk);
  __DSB();
  __ISB();
}
```

3.3.2 Linker Script Configuration

In the linker file, a new region corresponding to the TCM must be created so that relevant code and data can be linked to the right memory addresses.

For Atmel Studio/GCC compiler the linker file is a *.ld* file. Firstly ITCM and DTCM memory regions have to be defined, with their respective base addresses, list of attributes (read, write, execute etc.) and size:

```
/* ITCM and DTCM are 128KB each */
itcm (rwx)  : ORIGIN = 0x00000000, LENGTH = 0x00020000
dtcm (rw)   : ORIGIN = 0x20000000, LENGTH = 0x00020000
```

Then a section needs to be created to place the relevant code and data in ITCM and DTCM respectively. Below is an example from the Atmel Studio project given with this application note, in which FFT computation functions are placed in TCM:

```
SECTIONS
{
```

```
.code_TCM 0x00000000:
AT ( _itcm_lma )
{
_sitcm = .;
fft.o (.text.*)
fft.o (.rodata)
*(EXCLUDE_FILE (library/* signal.o main.o wdt.o) .text)
*(EXCLUDE_FILE (library/* signal.o main.o wdt.o) .text.*)
*(EXCLUDE_FILE (library/* signal.o main.o wdt.o) .rodata.*)
*(EXCLUDE_FILE (library/* signal.o main.o wdt.o) .fini*)
_eitcm = .;
} > itcm
.data_TCM 0x20000000:
{
_sdtcm = .;
fft.o (.data)
fft.o (.bss)
fft.o (COMMON)
*(EXCLUDE_FILE (library/* signal.o main.o wdt.o) .data)
*(EXCLUDE_FILE (library/* signal.o main.o wdt.o) .bss*)
_edtcm = .;
} > dtcm
.DTCM_stack :
{
. = ALIGN(8);
_sdtcm_stack = .;
. += STACK_SIZE;
_edtcm_stack = .;
} > dtcm
}
```

For more information on linker script concepts, you can refer to the below application note:
http://www.atmel.com/Images/doc32158.pdf.

For IAR users, linker files are in the *.icf* format. In this .icf file, two regions must be created for ITCM and DTCM (`ITCM_region` and `DTCM_region` in the below example) with their respective start address and size:

```
define symbol __ICFEDIT_region_ITCM_start__= 0x00000000;
define symbol __ICFEDIT_region_DTCM_start__= 0x20000000;
define symbol __ICFEDIT_size_itcm__= 0x20000;
define symbol __ICFEDIT_size_dtcm__= 0x20000;


define region ITCM_region = mem:[from __ICFEDIT_region_ITCM_start__ size
__ICFEDIT_size_itcm__];
define region DTCM_region = mem:[from __ICFEDIT_region_DTCM_start__ size
__ICFEDIT_size_dtcm__];
```

Then relevant functions and data are placed in TCM. The below example is from the IAR project coming with the application note:

```
place in ITCM_region    {
   readwrite object fft.o,
   readwrite object arm_cortexM7lfdp_math.lib,
   readwrite object dl7M_tln.a,
   readwrite object m7M_tlv.a,
   readwrite object rt7M_tl.a
};

place in DTCM_region   {
   zi object fft.o,
   zi object arm_cortexM7lfdp_math.lib,
   zi object dl7M_tln.a,
   zi object m7M_tlv.a,
   zi object rt7M_tl.a
};

initialize by copy      {
   object fft.o,
   object arm_cortexM7lfdp_math.lib,
   object dl7M_tln.a,
   object m7M_tlv.a,
   object rt7M_tl.a
};
```

For more information on IAR linker files, you can refer to the following application note:

http://supp.iar.com/FilesPublic/UPDINFO/005316/xlink.ENU.pdf.

3.3.3 Code Copy in TCM

When compiling an application code with Atmel Studio/GCC compiler, the code to move in TCM area must be copied manually. This is usually done during initialization, just before reaching the main function. The below code snippet shows how to do it:

```
/* copy code_TCM from flash to ITCM */
volatile char *dst = &_sitcm;
volatile char *src = &_itcm_lma;
while(dst < &_eitcm){
      *dst++ = *src++;
}
```

For IAR users, this copy loop is already implemented in the IAR cstartup file, so there is no need add it in the application step.

3.4 Optimize Pipeline Usage

In order to fully benefit from the dual-instruction issue capability of the Cortex-M7 pipeline, it is recommended to interleave load and store instructions as shown below:

```
Xn1 = pIn[0];
Xn2 = pIn[1];

Xn3 = pIn[2];
acc1 = b0 * Xn1 + d1;

Xn4 = pIn[3];
d1 = b1 * Xn1 + d2;

Xn5 = pIn[4];
d2 = b2 * Xn1;

Xn6 = pIn[5];
d1 += a1 * acc1;

Xn7 = pIn[6];
d2 += a2 * acc1;
```

Compilers are not able to perform such kind of optimization thus it must be done at application code level.

Note: The ARM CMSIS library for Cortex-M7, which is used on all software examples for SAM S70/E70/V7x, already implements such optimization mechanism.

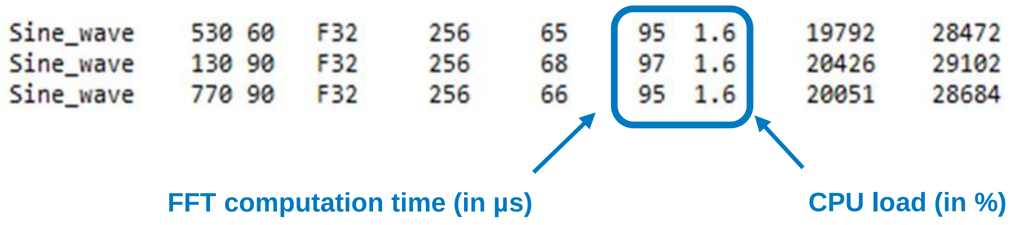
4 Software Example: FFT Computation Code Example

This part is an introduction of the code example provided with this application note, which allows to illustrate the different concepts previously detailed. This code example is provided for Atmel Studio 7 (build 7.0.567 or above), and needs the Atmel | SMART SAM V71 Xplained Ultra board.

This code example is based on the ARM CMSIS-DSP library version 1.4.3 (CMSIS version 3.20).

4.1 Introduction to the Example

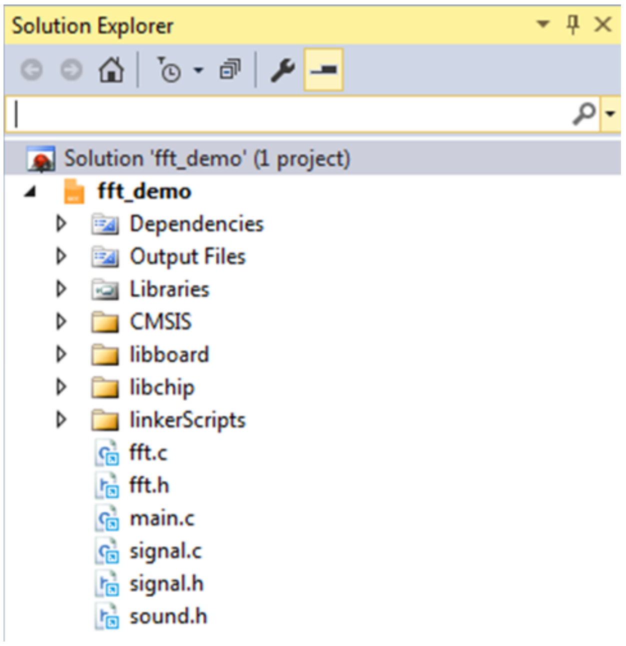
The purpose of the example is to generate a 256-point FFT of a sine wave using 32-bit floating-point data type (F32). FFT computation time and CPU load are displayed in real time on the DEBUG USB Virtual COM port:



```
Sine_wave    530 60    F32    256    65    95   1.6    19792    28472
Sine_wave    130 90    F32    256    68    97   1.6    20426    29102
Sine_wave    770 90    F32    256    66    95   1.6    20051    28684
```

As shown in the below screenshot, the project is made of three folders:

- **libchip**: provides the API for the different embedded peripherals
- **libboard**: provides the API for the different on-board components and board low-level initialization
- **linkerScripts**: contains the linker scripts



```
Solution 'fft_demo' (1 project)
  fft_demo
    Dependencies
    Output Files
    Libraries
    CMSIS
    libboard
    libchip
    linkerScripts
    fft.c
    fft.h
    main.c
    signal.c
    signal.h
    sound.h
```

The following source files are also noticeable:

- **main.c**: program entry-point. In this file you will find the initialization function of the device, and the sine wave generation function. The I- and D-cache enable functions can be found at the beginning of the *main* function (*SCB_EnableICache()* and *SCB_EnableDCache()* functions).
- **fft.c**: contains all functions related to the FFT computation process. This process calls the mathematics functions from the ARM CMSIS-DSP library (included in Atmel Studio).

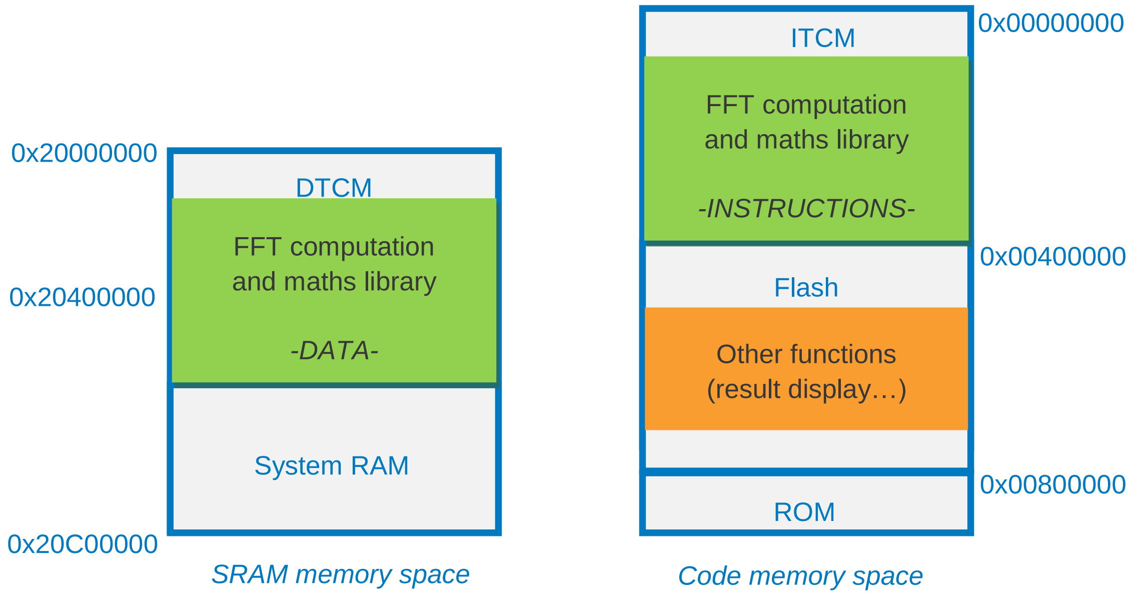
- **startup_sam.c** (in the **libboard** folder): contains the exception table and reset handler. The reset handler sets the vector table, and also configure the TCM (GPNVM bits) and enable it.
- **board_lowlevel.c** (in the **libboard** folder): Performs the low-level initialization of the chip, including EEFC (Flash Controller), MPU and master clock configuration. The code copy in TCM memory can be also found at the end of the *LowLevelInit()* function in this file.
- **samv71q21_xxx.ld** (in **linkerScripts** directory): program linker file. As explained previously its purpose is to describe how the sections in the input files should be mapped into the output file (.*hex* file), and to control the memory layout of the output file.

4.2 Usage

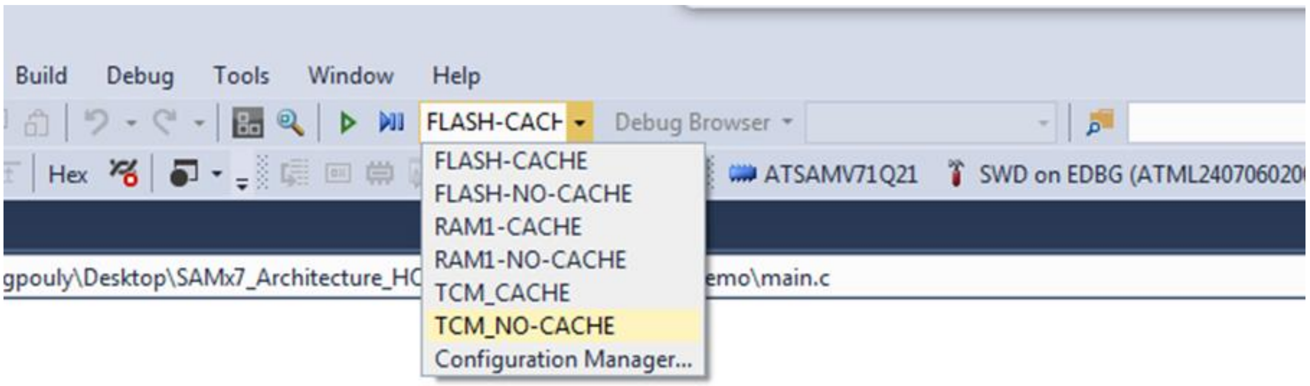
The project contains several configurations so that FFT computation process runs out of flash (with I-/D-cache enabled or not), System RAM or TCM.

Figure 4-1 shows the case with FFT computation is executed out of TCM:

Figure 4-1. FFT Computation Process Running out of TCM



The configuration is changed through the drop-down menu at the top of the tools bar:



- FLASH-CACHE executes the FFT computation process out of flash with I-/D-cache enabled
- FLASH-NO-CACHE executes the FFT computation process out of flash with I-/D-cache disabled
- RAM1-CACHE executes the FFT computation process out of System RAM with I-/D-cache enabled
- RAM1-NO-CACHE executes the FFT computation process out of System RAM with I-/D-cache disabled
- TCM-CACHE executes the FFT computation process out of TCM with I-/D-cache enabled
- TCM-NO-CACHE executes the FFT computation process out of TCM with I-/D-cache disabled

4.3 Results

As mentioned in introduction the code example is based on the ARM CMSIS-DSP library version 1.4.3 (CMSIS version 3.20) and running on the SAM V71 Xplained Ultra board.

The results to expect are the following (FFT computation time in us, CPU load in %):

	Flash	System RAM	TCM
I/D cache disabled	358µs 6.1%	341µs 5.8%	79µs 1.3%
I/D cache enabled	102µs 1.7%	95µs 1.5%	79µs 1.3%

The maximum performance level is achieved when FFT computation code is running out of TCM, thanks to the direct access to the Cortex-M7 processor at the processor clock speed (300MHz).

The results when running out of flash with cache enabled are pretty close because instruction and data in cache are also fetched at processor clock speed. As explained previously the main difference between code execution out of cache and TCM is determinism: TCM is more suitable when performing real-time tasks because it is fully deterministic. The other difference is that TCM management mechanism is simpler: unlike cache, there are no maintenance operations to perform, such as invalidate, clean etc.

5 Conclusion

This application note presented the new features introduced by the Cortex-M7 core and how it was integrated in the SAM S70/E70/V7x devices. We also had a review of the SAM S70/E70/V7x architecture to have a better understanding on why these devices are perfectly suited for highly performance demanding application. Finally a concrete example was given to illustrate how to enable and properly use these specific features.

6 Suggested Readings

- Cortex-M7 Processor Technical Reference Manual:
  http://infocenter.arm.com/help/topic/com.arm.doc.ddi0489b/DDI0489B_cortex_m7_trm.pdf
- Atmel | SMART SAM S70 Product Datasheet:
  http://www.atmel.com/Images/Atmel-11242-32-bit-Cortex-M7-Microcontroller-SAM-S70Q-SAM-S70N-SAM-S70J_Datasheet.pdf
- Atmel | SMART SAM E70 Product Datasheet:
  http://www.atmel.com/Images/Atmel-11296-32-bit-Cortex-M7-Microcontroller-SAM-E70Q-SAM-E70N-SAM-E70J_Datasheet.pdf
- Using the MPU on Atmel Cortex-M3 / Cortex-M4 based Microcontrollers:
  http://www.atmel.com/Images/Atmel-42128-AT02346-Using-the-MPU-on-Atmel-Cortex-M3-M4-based-Microcontroller_Application-Note.pdf
- ARM Cortex-M Programming Guide to Memory Barrier Instructions:
  http://infocenter.arm.com/help/topic/com.arm.doc.dai0321a/DAI0321A_programming_guide_memory_barriers_for_m_profile.pdf

7 Revision History

Doc Rev.	Date	Comments
44047B	03/2016	Code example updated to Atmel Studio 7.
44047A	06/2015	Initial document release.

Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**