AN4515
Application note

Using Batch Acquisition Mode (BAM) to maximize power efficiency on STM32F410, STM32F411, STM32F412, STM32F413 microcontroller lines

Introduction

The STM32F410, STM32F411, STM32F412 and STM32F413 lines are part of the STM32 Dynamic Efficiency™ microcontrollers. These devices are the entry level to the high-performance STM32F4 Series and offer the best balance of dynamic power consumption and processing performance, while achieving outstanding low-power consumption in Run, Sleep and Stop modes.

With the new Batch Acquisition Mode (BAM) which allows optimizing the power consumption for data batching, the STM32F410, STM32F411, STM32F412 and STM32F413 microcontroller lines bring the Dynamic Efficiency to a new level.

This application note guides the user to achieve the expected power consumptions on the STM32F410, STM32F411, STM32F412 and STM32F413 microcontrollers through an application case. It also provides an example of how to implement the BAM.

This application note is provided with X-CUBE-BAM firmware package.

Contents

1 Application Overview 5
1.1 Hardware high level description 5
1.2 Low-power modes 6
1.3 Batch Acquisition Mode (BAM) 6
1.3.1 Principle 6
1.3.2 BAM use case 6
1.3.3 How to implement BAM 7
1.3.4 How to execute code from RAM using Keil® MDK-ARM™ toolchain . . 11
1.3.5 How to execute code from RAM using IAR-EWARM toolchain 13
1.3.6 How to execute code from RAM using AC6-SW4STM32 toolchain . . . 16

2 Application note use case 19
2.1 Block diagram 19
2.2 Peripherals used in STM32F410, STM32F411, STM32F412 and STM32F413 lines 20
2.3 Functional description 21

3 Application current consumption 24
3.1 Hardware requirements 24
3.2 Software settings 25
3.3 Measured current consumption 25

4 Conclusion 26

5 Revision history 27

List of tables

Table 1. Summary of methods 10
Table 2. Master and sensor board connection 24
Table 3. Current consumption example 25
Table 4. Document revision history 27

List of figures

Figure 1. Application high level block diagram 5
Figure 2. Batch Acquisition Mode use case 7
Figure 3. Executing ISRs from RAM 8
Figure 4. Executing ISRs from Flash memory 9
Figure 5. Using events to wake up the CPU 10
Figure 6. MDK-ARM file placement 11
Figure 7. MDK-ARM scatter file 12
Figure 8. MDK-ARM Options menu 12
Figure 9. Update of EWARM linker 13
Figure 10. Update of EWARM startup file to handle an interrupt 15
Figure 11. GNU linker updated 16
Figure 12. GNU Linker section definition 17
Figure 13. ".RAMVectorTable" section 17
Figure 14. Declaration of globale variable located in ".RAMVectorTable" 17
Figure 15. Vector table relocation 18
Figure 16. Startup file update 18
Figure 17. Use case block diagram 19
Figure 18. Use case state machine 21
Figure 19. Startup and Configuration menu 22
Figure 20. Wait MEMS movement while CPU is Sleep mode and Flash memory stopped 23
Figure 21. Log example when configuration 4 is selected 23
Figure 22. Hardware environment 24

1 Application Overview

This section gives an overview of the application note use case, details how the user can implement the BAM and describes each low-power mode involved in the use case operation.

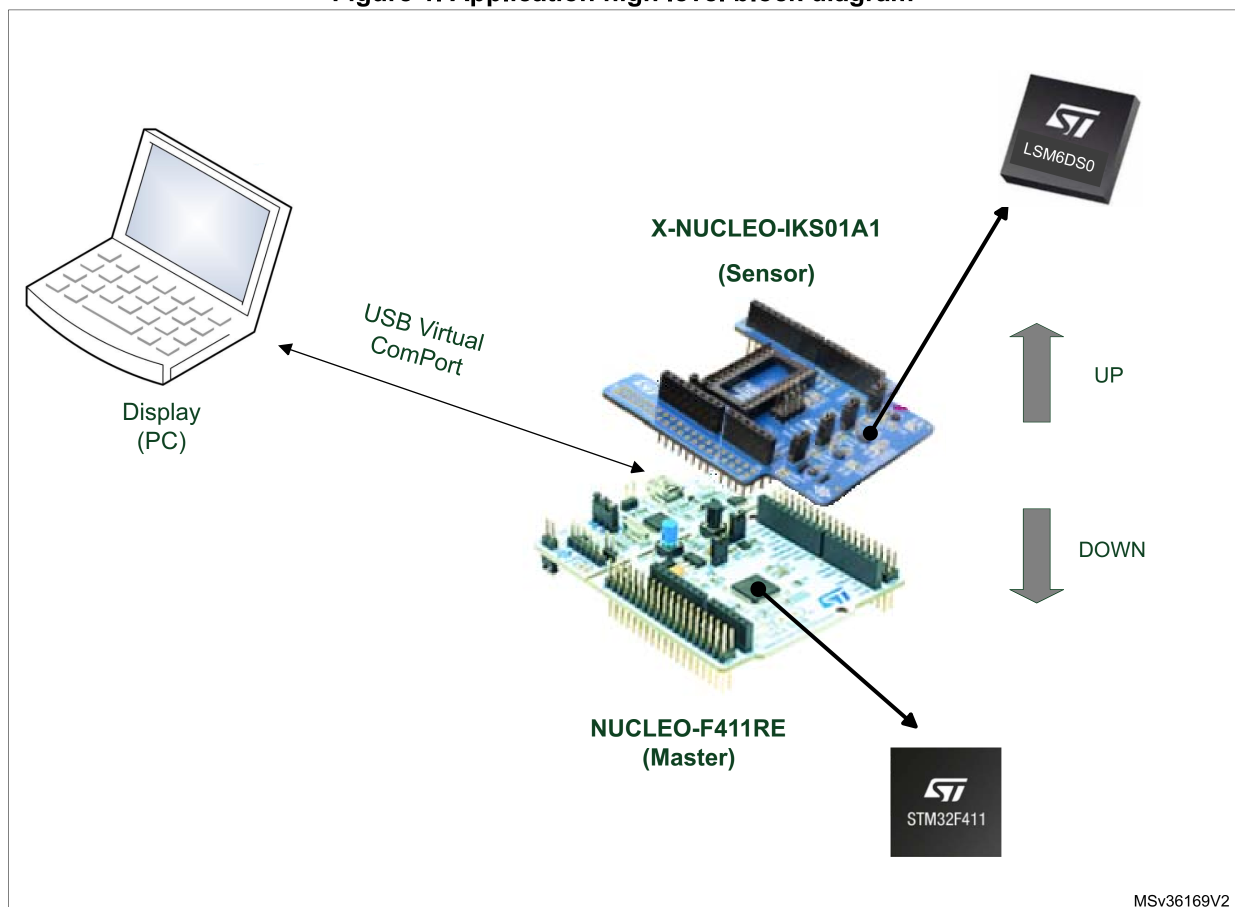
1.1 Hardware high level description

This document first explains how to assess the power consumption in different low-power modes (Standby, Sleep, Stop and Low-power run) and then highlights the significant power reduction achieved by using the BAM to transfer sensor (LSM6DS0 MEMS) data direction (UP or DOWN) to a master STM32F410/STM32F411/412/413xx microcontroller.

The data sent by the sensor is received via the I2C interface and transferred by DMA to the RAM. After processing data, the DMA sends the data direction to the UART. It is then displayed via the USB virtual comport.

Figure 1 shows a high-level block diagram of the application.

Figure 1. Application high level block diagram



1. The above block diagram also applies to STM32F410, STM32F412 and STM32F413 lines.

1.2 Low-power modes

This section describes the different low-power modes used in the application.

- **Low-power run mode**
  The CPU and some of the peripherals are running. To further reduce the power consumption, unused GPIOs configured as analog pins and peripherals are disabled.
- **Sleep mode**
  In this mode only the CPU is stopped and the peripherals kept running. To reduce the power consumption the Flash memory can be stopped before entering Sleep mode. Peripherals can wake up the CPU when an interrupt occurs.
- **Stop mode**
  The CPU is in DeepSleep mode. All peripherals except RTC are disabled. Exiting Stop mode is done by issuing an interrupt.
- **Standby mode**
  In this mode the power is only maintained for the RTC registers. The device is woken by a rising edge on the WKUP pin that generates a system reset.

Note: For more details about the low-power modes refer to the product datasheets.

1.3 Batch Acquisition Mode (BAM)

1.3.1 Principle

The Batch Acquisition Mode (BAM) optimizes the power consumption for data batching.

It allows exchanging batches of data through communication peripherals while the rest of the device (including the CPU) is in low-power mode:

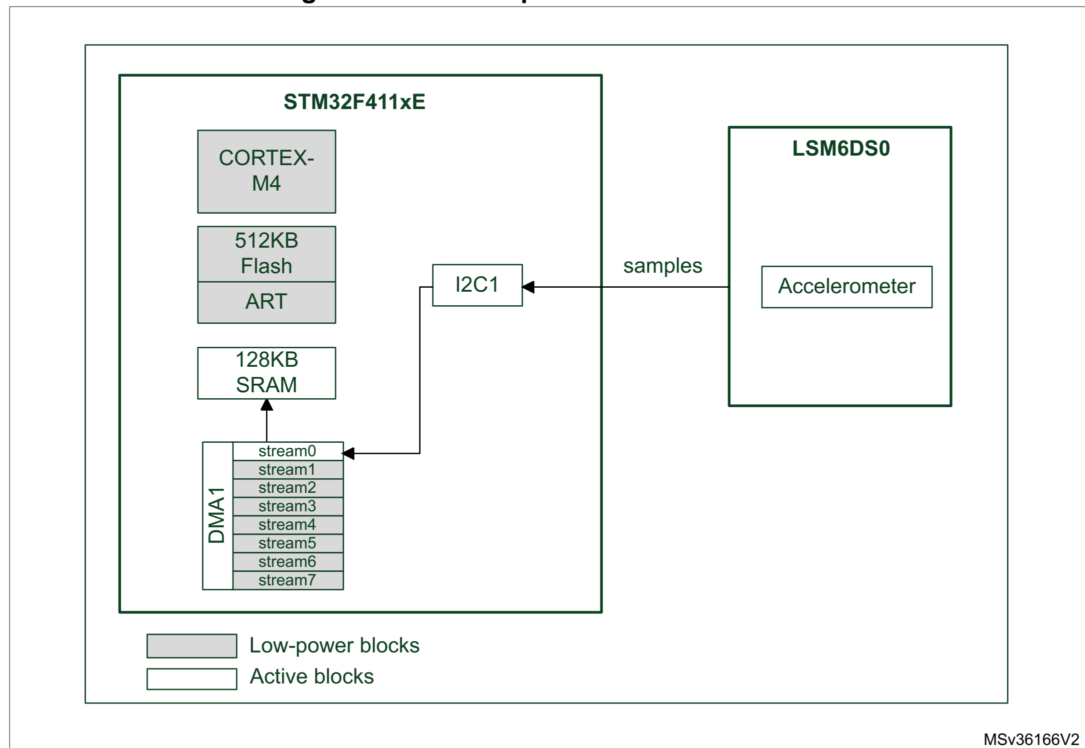
- Only the needed DMA streams are enabled and running to transfer data from communication interfaces to internal RAM.
- Routine execution from RAM allows switching off the Flash memory and stopping the Flash Interface clock.
- The MCU core is put in Sleep mode waiting for an interrupt/event to wake up.

1.3.2 BAM use case

Figure 2 shows one part of the application note use case block diagram. It describes the different peripherals involved in the BAM and mentions their power state (low-power or active).

In fact, during data reception, only the DMA stream0, I2C1 and SRAM are active. The MCU core is in Sleep mode and the Flash memory is stopped until the DMA transfer is complete.

Figure 2. Batch Acquisition Mode use case



1. This use case also applies to STM32F410, STM32F412 and STM32F413 lines.

1.3.3 How to implement BAM

When the BAM is enabled, the Flash memory is switched off by setting the FMSSR and FISSR bits in PWR_CR register. These bits cannot be set while executing the program from the Flash memory. This can be done through a specific routine executed from RAM (see *Figure 3*, *Figure 4* and *Figure 5* for a description).

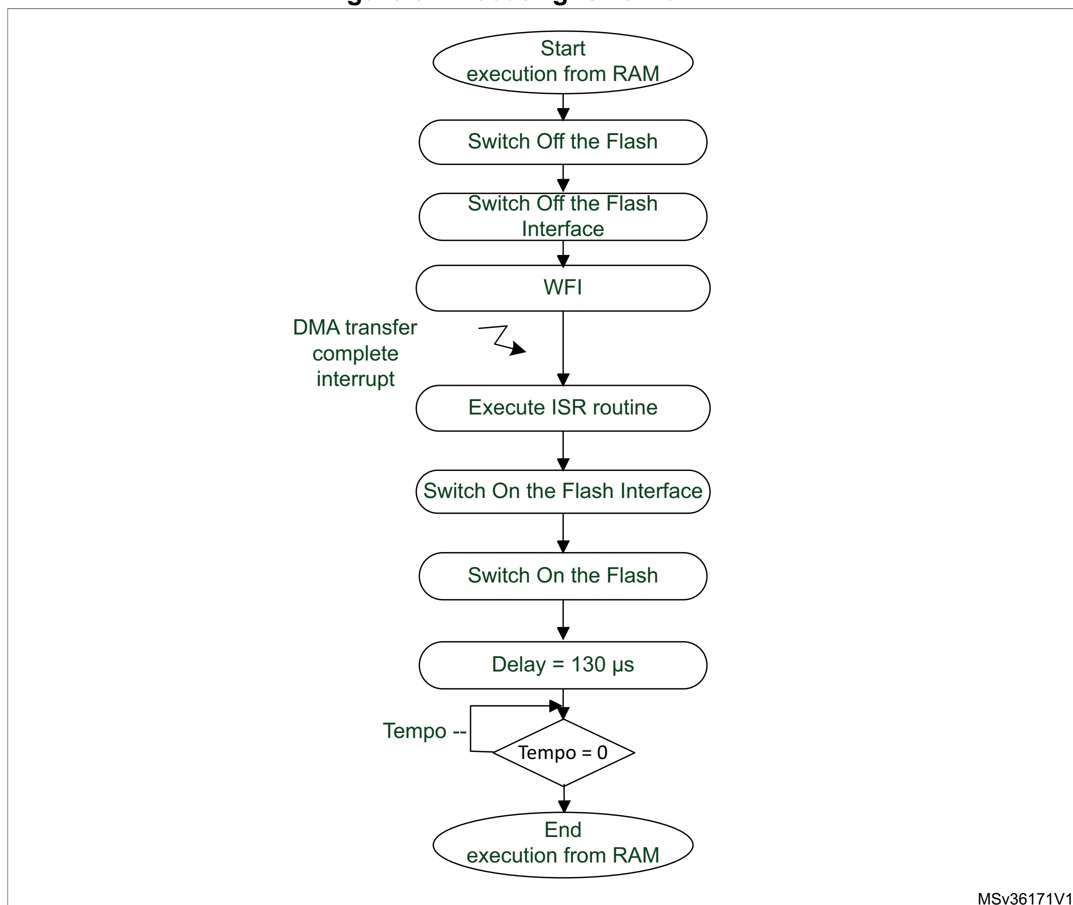
Three methods can be used to implement the BAM. Each approach depends on user application:

- Executing interrupts from RAM
- Executing interrupts from Flash memory
- Using events to wake up the CPU

Executing interrupts from RAM

This is the approach used in the application note use case (see *Figure 3*). The device is woken up from Sleep by an interrupt. The user application has therefore to store the required ISRs and the vector table in the RAM so that it is immediately executed from the RAM when the interrupt occurs.

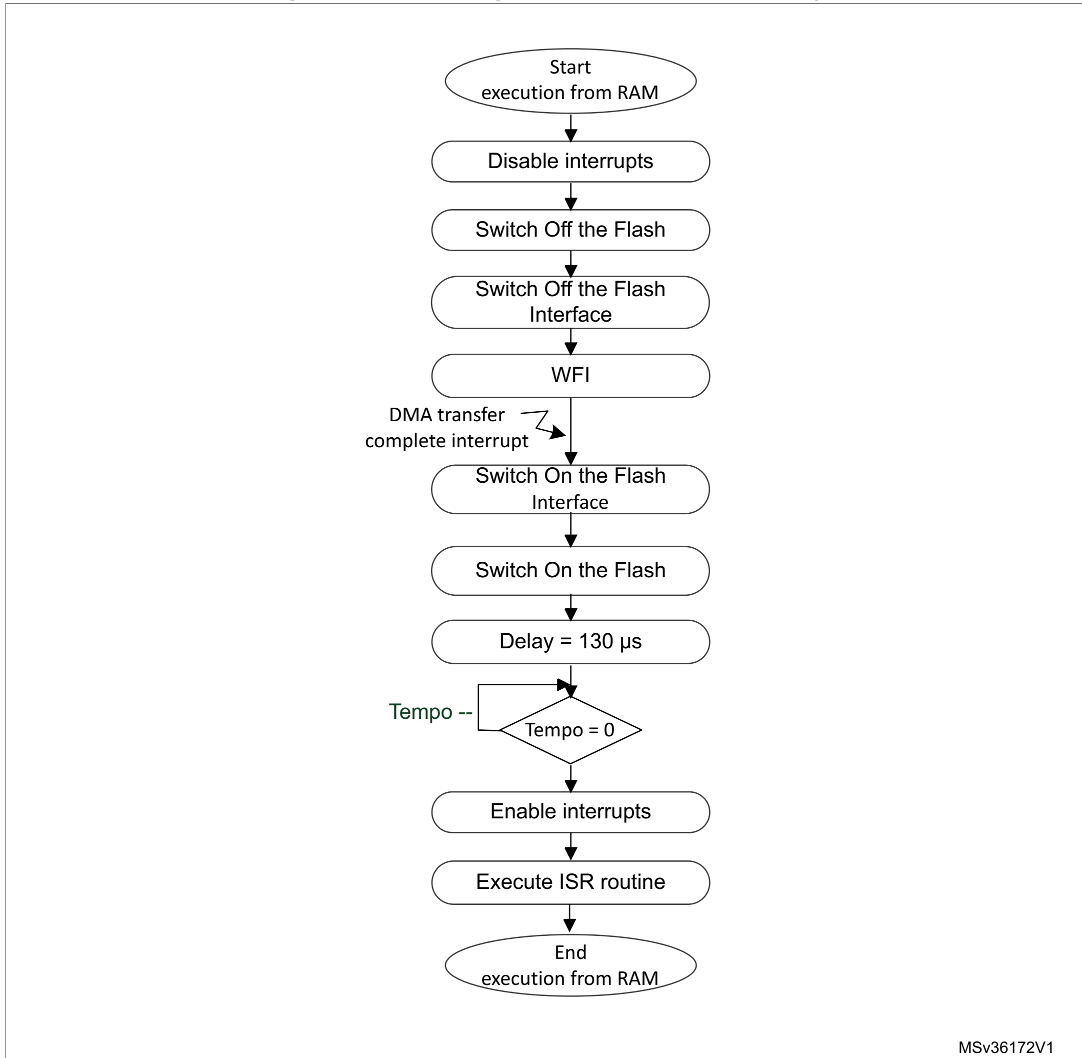
Figure 3. Executing ISRs from RAM



Executing Interrupts from Flash memory

In this approach the Flash memory is stopped and all the interrupts must be disabled until the Flash memory is enabled again (see *Figure 4*).

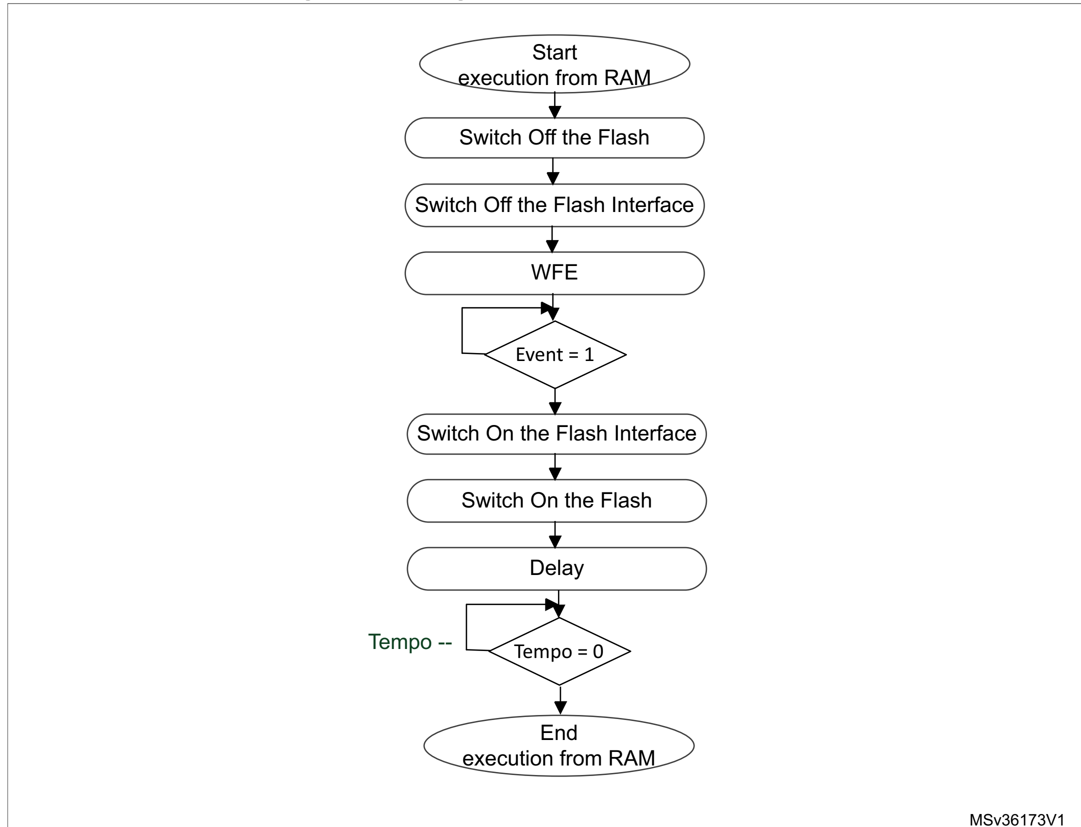
Figure 4. Executing ISRs from Flash memory



Using events to wakeup CPU

The method uses events instead of interrupts to wake up the CPU from Sleep mode (see *Figure 5*).

Figure 5. Using events to wake up the CPU



Summary of BAM implementation methods

Table 1. Summary of methods

Approach	Strength	Weakness
Executing ISRs from RAM	No latency in interrupt execution	FW little bit complex to implement
Executing ISRs from Flash memory	Easy firmware implementation	Latency in interrupt execution
Using events to wake up the CPU		Interrupts are not executed since the Flash memory is stopped

1.3.4 How to execute code from RAM using Keil® MDK-ARM™ toolchain

This section gives an overview of the steps required to execute part of the application code from RAM using the Keil MDK-ARM toolchain.

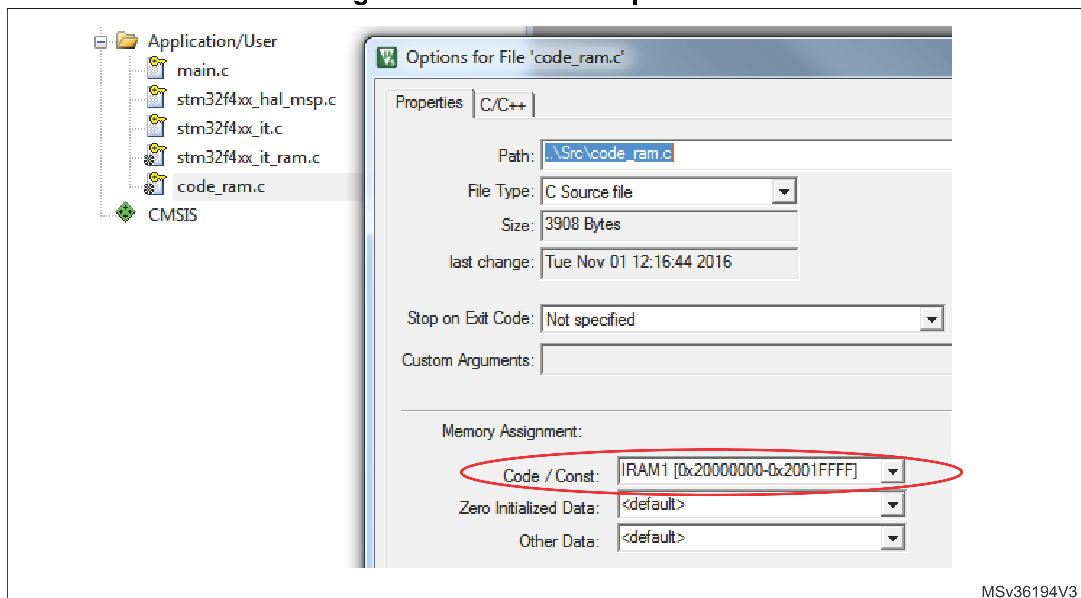
Executing a source file from RAM

Executing a source file from RAM means that all the functions declared in this file will be executed from RAM area.

Follow the steps below to execute a file from RAM (see *Figure 6*):

1. Right click the file to place it in RAM and select **Options.**
2. Select the RAM area in the **Memory Assignment** menu.

Figure 6. MDK-ARM file placement



Executing an interrupt from RAM

The following steps are required to execute an interrupt handler from RAM:

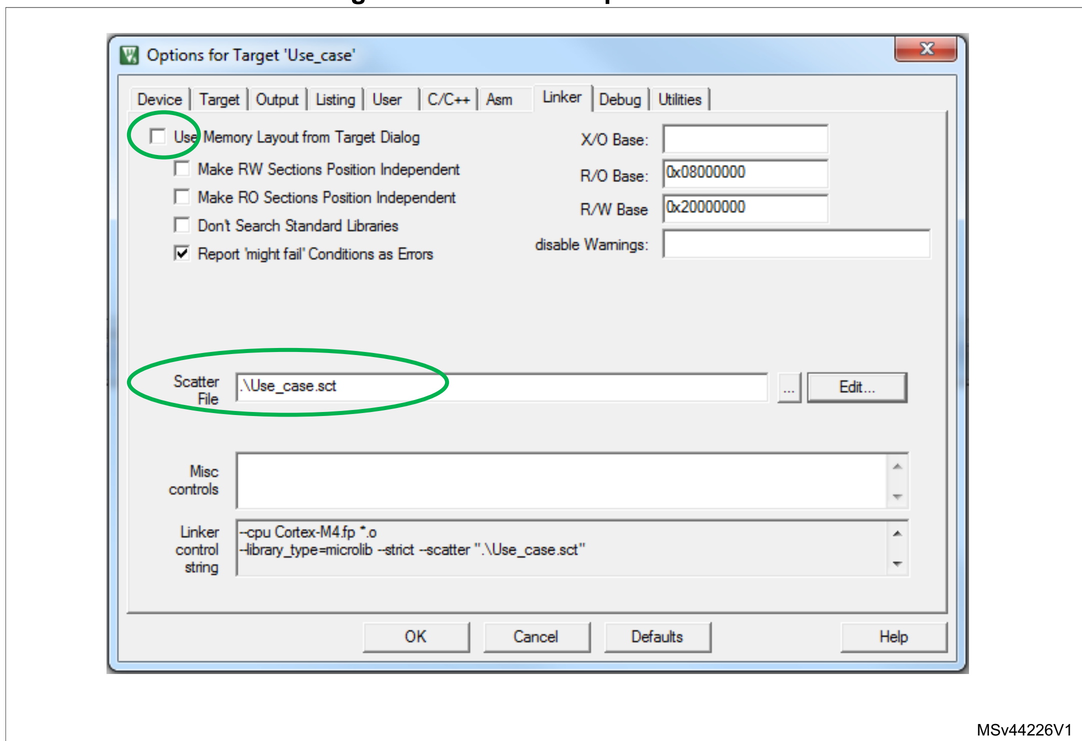
1. Make a second vector table and save it in a new file (startup_stm32f411xe_ram.s) that will be saved in RAM.
2. Place the interrupt handler to be executed from RAM in a new file named stm32f4xx_it_ram.c.
3. Place the required files in the right RAM area in the scatter file (see *Figure 7*).

Figure 7. MDK-ARM scatter file

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x08000000 0x00080000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00080000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x20000000 0x00020000  {  ; RW data
  *.o (RESET_ram, +First)
    startup_stm32f411xe_ram.o (+RO)
    code_ram.o (+RO +RW)
    stm32f4xx_it_ram.o (+RO +RW)
   .ANY (+RW +ZI)
  }
}
```

4. Refer to the modified scatter file for the project options (see *Figure 8*).

Figure 8. MDK-ARM Options menu



1.3.5 How to execute code from RAM using IAR-EWARM toolchain

Executing code from RAM

This operation requires a new RAM0 section to be defined in the linker file (.icf) (see *Figure 9*). The required steps are the following:

1. Define the address area for RAM0 by indicating the start and end addresses.
2. Tell the linker to copy at startup the section named ‘ram0’ from Flash memory to RAM0.
3. Indicate to the linker that ram0 code section should be placed in the RAM0 area.

Figure 9. Update of EWARM linker

```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x08000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x0807FFFF;

define symbol __ICFEDIT_region_RAM0_start__  = 0x20000000;
define symbol __ICFEDIT_region_RAM0_end__    = 0x20001FFF;

define symbol __ICFEDIT_region_RAM_start__ = 0x20002000;
define symbol __ICFEDIT_region_RAM_end__ = 0x2001FFFF;

/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x800;
define symbol __ICFEDIT_size_heap__   = 0x400;
/**** End of ICF editor section. ###ICF###*/

define symbol RAM0_intvec_start = 0x20000000;

define memory mem with size = 4G;
define region ROM_region      = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__];
define region RAM_region      = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__];

define region RAM0_region      = mem:[from __ICFEDIT_region_RAM0_start__    to __ICFEDIT_region_RAM0_end__];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy { readwrite, section .ram0, section .intvec_RAM0, ro object stm32f4xx_it_ram.o };
do not initialize  { section .noinit };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place at address mem:RAM0_intvec_start { section .intvec_RAM0 };

place in ROM_region   { readonly };
place in RAM_region   { readwrite,
                        block CSTACK, block HEAP };
place in RAM0_region   { section .ram0 };
```

Executing a source file from RAM

To execute a source file from RAM, use the file Options window:

1. Select **Options** from the displayed menu.
2. Select **C/C++ Compiler**
3. Check **override inherited settings** from the displayed window.
4. Select the **output** tab and type the name of the section already defined in the linker file (**.ram0** in this example) in the **Code section name** field.

Executing an interrupt from RAM

The steps required to execute an interrupt handler from RAM0 are the following:

1. Update the linker file (.icf) (see *Figure 9*).
   a) Define the address where the second vector table will be located: 0x2000 0000.
   b) Tell the linker to copy the section named .intvec_RAM0 at startup.
2. Update the startup file (see *Figure 10*)
3. Place the interrupt handlers to be executed in RAM0 in the new **stm32F4xx_it_ram.c** as described in *Section : Executing a source file from RAM*.
4. Remap the vector table to RAM0. To do this, modify the VTOR register in SystemInit function as following:

   SCB->VTOR = 0x2000 0000 | VECT_TAB_OFFSET

Figure 10. Update of EWARM startup file to handle an interrupt

```
        MODULE  ?cstartup

        ;; Forward declaration of sections.
        SECTION CSTACK:DATA:NOROOT(3)

        SECTION .intvec:CODE:NOROOT(2)

        EXTERN  __iar_program_start
        EXTERN  SystemInit
        PUBLIC  __vector_table

        DATA

__vector_table
        DCD     sfe(CSTACK)
        DCD     Reset_Handler             ; Reset Handler

        SECTION .intvec_RAM0:CODE:ROOT(2)

        PUBLIC  __vector_table_RAM0

        DATA

__vector_table_RAM0
        DCD     sfe(CSTACK)
        DCD     Reset_Handler             ; Reset Handler

        DCD     NMI_Handler               ; NMI Handler
        DCD     HardFault_Handler         ; Hard Fault Handler
        DCD     MemManage_Handler         ; MPU Fault Handler
        DCD     BusFault_Handler          ; Bus Fault Handler
        DCD     UsageFault_Handler        ; Usage Fault Handler
        DCD     0                         ; Reserved
        DCD     0                         ; Reserved
```

1.3.6 How to execute code from RAM using AC6-SW4STM32 toolchain

This section explains how to use these features to execute the code from RAM.

Executing a function or an interrupt handler from RAM

The steps required to execute a function or an interrupt handler from RAM0 are the following:

1. Declare two new memory regions (SRAM and RAM0) in the linker file (.ld) by defining the start address and the size. (*See Figure 11*)
   – The SRAM has a read / write / execute permission, 1 Kbyte size and start address = 0x20000000.
     SRAM: is the area where the vector table will be located.
   – The RAM0 has a read / write / execute permission, 7 Kbyte size and start address = 0x20000400.
     RAM0 is the area where the source files will be executed.

Figure 11. GNU linker updated

```
/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = 0x20020000;    /* end of RAM */
/* Generate a link error if heap and stack don't fit into RAM */
_Min_Heap_Size = 0x400;      /* required amount of heap  */
_Min_Stack_Size = 0x800; /* required amount of stack */

/* Specify the memory areas */
MEMORY
{
FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 512K

SRAM (xrw)      : ORIGIN = 0x20000000, LENGTH = 1K

RAM0 (xrw)      : ORIGIN = 0x20000400, LENGTH = 7K

RAM (xrw)      : ORIGIN = 0x20002000, LENGTH = 128K
}
```

2. Define the new section in the linker file as shown in *Figure 12*
   - Define the symbol '_siRAM0Code' to store the load address in the Flash memory of the code intended to be executed in SRAM.
   - The symbols '_sRAM0Code' and '_eRAM0Code' are defined for the start and end addresses of the collection of '. RAM0Code' sections.

     → These symbols are used by the startup file (See *Figure 16*).
   - Place the files in RAM0 as shown below:

Figure 12. GNU Linker section definition

```
 /* used by the startup to initialize code from flash to RAM0 */
 _siRAM0Code = LOADADDR(.RAM0Code);
.RAM0Code :
{
  . = ALIGN(4);
  _sRAM0Code = .;        /* create a global symbol at RAM0Code start */

  *code_ram.o (.text .text* .rodata .rodata* .data .data* .bss .bss*)
  *stm32f4xx_it_ram.o (.text .text* .rodata .rodata* .data .data* .bss .bss*)

  . = ALIGN(4);
  _eRAM0Code = .;        /* define a global symbol at RAM0Code end */
} >RAM0 AT> FLASH
```

3. Place ".RAMVectorTable" section at the SRAM region in the linker file:

Figure 13. ".RAMVectorTable" section

```
.ARM.attributes 0 : { *(.ARM.attributes) }

.RAMVectorTable : {*(.RAMVectorTable)} >SRAM AT> FLASH
```

4. Update main.c file by adding the macro function mentioned below:
   - Declare a global array placed in ".RAMVectorTable" section (see *Figure 14*)

     This array holds the vector table.

Figure 14. Declaration of globale variable located in ".RAMVectorTable"

```
#ifdef __GNUC__
__IO uint32_t VectorTable[101] __attribute__((section(".RAMVectorTable")));
#endif
```

– Copy the vector table to the global array VectorTable[] (see *Figure 15*).
  This step ensures that the vector table is placed in the internal SRAM at 0x20000000

Figure 15. Vector table relocation

```
#ifdef __GNUC__
uint32_t index=0;
/* Relocate by software the vector table to the internal SRAM at 0x20000000 ***/
        /* Copy the vector table from the Flash (mapped at the base of the application
        load address 0x08004000) to the base address of the SRAM at 0x20000000. */
        for (index = 0; index < 101; index++)
        {
        VectorTable[index] = *(__IO uint32_t*)(0x08000000 + (index<<2));
        }
#endif
```

5. Relocate the vector table to RAM0 by modifying the VTOR register in the SystemInit function as following: SCB->VTOR = 0x2000 0000 | VECT_TAB_OFFSET
6. Modify the startup file to initialize data to be placed in RAM0 at the startup time. (See *Figure 16*)

Figure 16. Startup file update

```
/* Call the clock system intitialization function.*/
  bl  SystemInit

  /* Copy the data segment initializers from flash to RAM0 */
  movs  r1, #0
  b LoopCopyRAM0CodeInit

CopyRAM0CodeInit:
ldr r3, =_siRAM0Code
ldr r3, [r3, r1]
str r3, [r0, r1]
adds r1, r1, #4
LoopCopyRAM0CodeInit:
ldr r0, =_sRAM0Code
ldr r3, =_eRAM0Code
adds r2, r0, r1
cmp r2, r3
bcc CopyRAM0CodeInit

/* Call static constructors */
        bl __libc_init_array
/* Call the application's entry point.*/
  bl  main
  bx  lr
.size  Reset_Handler, .-Reset_Handler
```

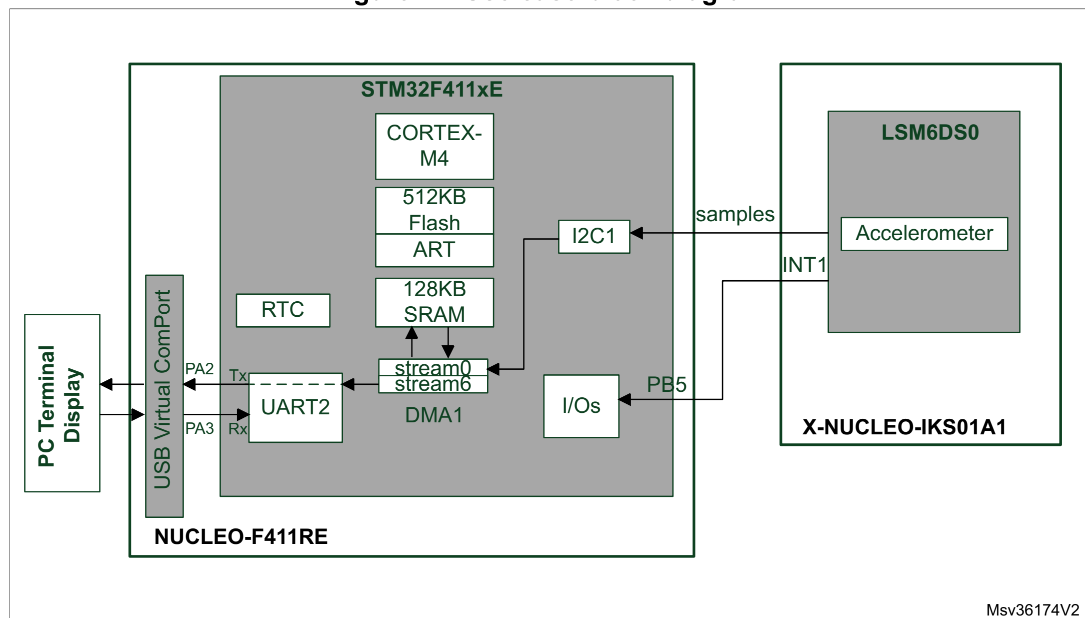
2 Application note use case

This section describes the case developed in this application note through a block diagram and a state machine highlighting the different peripherals and the low-power modes that are used.

2.1 Block diagram

Figure 17 describes the different modules involved in the use case.

Figure 17. Use case block diagram



1. This use case also applies to STM32F410, STM32F412 and STM32F413 lines.

2.2 Peripherals used in STM32F410, STM32F411, STM32F412 and STM32F413 lines

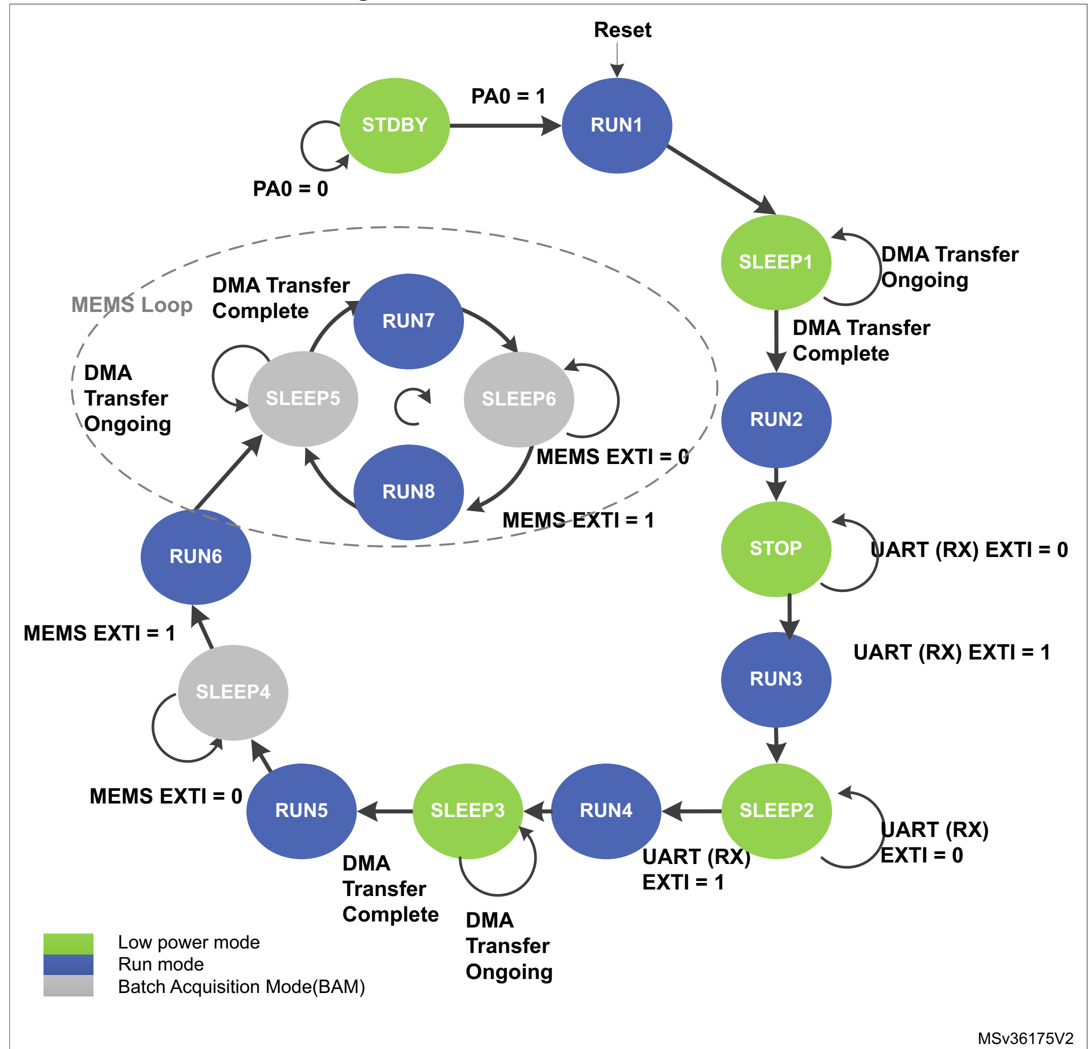
The application use case makes use of the below STM32F410/STM32F411/STM32F412/STM32F413 peripherals:

- **Clocks**
  Two clocks are used in this application:
  - HSI: 16 MHz system clock source
  - LSE: 32.768 kHz low-speed external crystal which drives the RTC clock
- **SysTick timer**
  This timer is used for waiting loops or to generate timeouts (delays).
- **GPIOs**
  - PA0: used as user pushbutton and wakeup pin from Standby mode
  - PA3: set as a pin with interrupt on USART2_RX input to wake up from Stop mode
  - PB5: set as an input with interrupt capability connected to the MEMS interrupt pin on X-NUCLEO-IKS01A1. This pin is used to wake up the CPU from sleep mode.
  - PB8(I2C1_SCL) and PB9(I2C1_SDA) connected to MEMS PB6 and PB9
  - When the microcontroller is in low-power mode, unused I/Os are placed in analog input mode to reduce the power consumption.
- **DMA1**
  The DMA1 is used to transfer data while the CPU is in Sleep mode.
  - Stream0: enabled for receiving data from I2C1
  - Stream6: enabled for transmitting data to USART2
  - Unused streams are disabled
- **RTC**
  The RTC is an independent timer/counter that provides a calendar with subseconds, seconds and minutes to generate timestamp events. It uses the LSE clock.
- **I2C1**
  The I2C1 interface receives data from the sensor at a speed up to 400 KHz.
- **USART2**
  The USART2 is used to transfer messages to USB virtual comport through the DMA.
  The USART2 is configured as follow:
  - Word Length = 8 bits
  - Stop Bit = one Stop bit
  - Parity = without parity
  - Baud rate = 9600 bauds
  - Hardware flow control disabled (RTS and CTS signals)
- **RAM**
  A specific routine is executed from RAM when the Flash memory is switched off.

2.3 Functional description

The principle of this use case is summarized in the state machine shown in *Figure 18*.

Figure 18. Use case state machine



The main steps of the application use case are the following:

1. The master and the sensor are powered on. By default the system is in **STDBY** state (Standby mode).
2. The user starts running the application by pushing the user button on the Nucleo board. The CPU goes to **SLEEP1** state while a DMA transfers startup message is displayed via USART2 and USB virtual comport.
3. When DMA transfers are complete, the CPU wakes up from **SLEEP1** and enters **STOP** waiting for the user to read the configuration menu below (see *Figure 19*):

   1: Flash active while CPU in sleep: Flash memory is running

   ***2:* Stop Flash while CPU in sleep:** this configuration brings out the BAM feature.

   ***3:* Enable Time Stamp and Flash active while CPU in sleep:** this option is used to track the different application state consumptions with using the period of each state displayed in timestamp log.

   4: Enable Time Stamp and stop Flash while CPU in sleep: this option allows benefiting both from the second and third configurations.

Figure 19. Startup and Configuration menu

```
WakeUp from STDBY ...
Enter to SLEEP1 ...


##########################################################
#                                                        #
# AN4515-Dynamic_Low_Power_Efficiency_In_The_STM32F411 #
#                                                        #
##########################################################


************** Configuration Menu *********************

1: Flash active while CPU in sleep
2: Stop Flash while CPU in sleep
3: Enable Time Stamp and Flash active while CPU in sleep
4: Enable Time Stamp and stop Flash while CPU in sleep

*******************************************************

WakeUp from SLEEP1 and enter to STOP ...

Read configuration menu and press any key board to wakeup
from STOP
>>
```

4. The system wakes up from Stop mode and enters **SLEEP2** state waiting for the user to select the desired configuration.
5. Once the configuration is selected, an external interrupt on USART2_RX wakes up the CPU from Sleep mode.
6. The CPU enters Sleep mode again (**SLEEP3** state) while a DMA transfers message is sent via USART2.
7. After a DMA transfer complete interrupt has been received, the CPU configures the MEMS and enters **SLEEP4** state waiting for an accelerometer movement (UP or DOWN direction). The DMA sends the ***### Move MEMS UP or DOWN ###*** message during Sleep mode.

Figure 20. Wait MEMS movement while CPU is Sleep mode and Flash memory stopped

```
Read configuration menu and press any key board to wakeup
from STOP
>>
WakeUp from STOP and enter to SLEEP2...


Select Your Configuration ...
>> 4

Configuration Selected Correctly
WakeUp from SLEEP2 ...
ENTER to SLEEP3 ...
WakeUp from SLEEP3 ...

Enter to SLEEP4 ...

### Move MEMS UP or DOWN ###
```

8. After the MEMS has moved, the CPU wakes up from Sleep mode, configures the I2C interface and DMA to transfer samples coming from the MEMS. The DMA then starts sending data to the RAM while the Flash memory is stopped and the CPU goes back to Sleep mode (**SLEEP5** state).
9. Once the DMA data transfer is complete, an interrupt wakes up the CPU to handle these data before it goes back to Sleep mode ((**SLEEP6** state)).
10. While the CPU is in Sleep mode (**SLEEP6** state) and the Flash memory is stopped, the DMA sends a complete log to the USART2 including MEMS direction and the timestamp (see *Figure 21*).

Figure 21. Log example when configuration 4 is selected

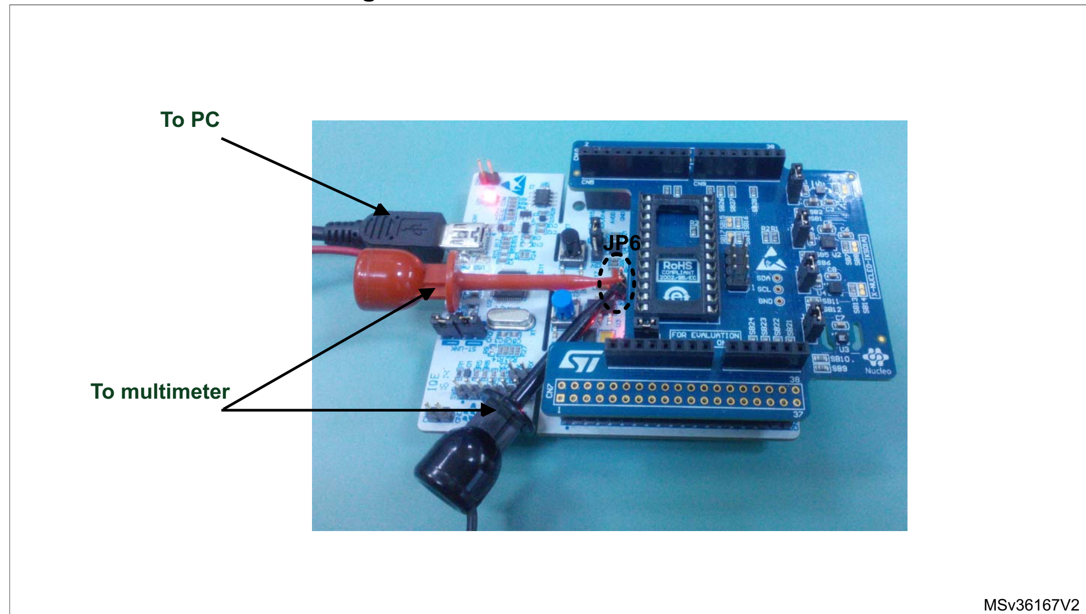
```
### Move MEMS UP or DOWN ###

************** Direction Display ***************
UP Direction

************** TimeStamp Display **************

00:00:00:000: Enter STDBY
00:00:02:145: WakeUp from STDBY
00:00:02:146: Enter to SLEEP1
00:00:02:904: WakeUp from SLEEP1
00:00:02:904: Enter to STOP
00:00:04:179: WakeUp from STOP
00:00:04:180: Enter to SLEEP2
00:00:04:982: WakeUp from SLEEP2
00:00:05:475: Enter to SLEEP3
00:00:05:552: WakeUp from SLEEP3
00:00:05:554: Enter to SLEEP4
00:00:09:078: WakeUp from SLEEP4
00:00:09:078: Enter to SLEEP5
00:00:09:079: Wakeup from SLEEP5
00:00:09:080: Enter to SLEEP6


## Move MEMS ##
```

3 Application current consumption

3.1 Hardware requirements

Before running the application and measuring the current consumption, connect your PC to the NUCLEO board as shown in *Figure 22*.

Figure 22. Hardware environment



The hardware required to run the application are the following:

- Windows® PC
- NUCLEO and X-NUCLEO-IKS01A1 boards
- One mini USB cable to power on the NUCLEO board and to connect the NUCLEO embedded ST-LINK for debugging and programming
- A multimeter to measure the current consumption.

Table 2. Master and sensor board connection

NUCLEO	X-NUCLEO-IKS01A1
I2C SCL (PB8)	I2C SCL
I2C SDA (PB9)	I2C SDA
PB5	LSM6DS0_INT1
GND	GND

3.2 Software settings

The measures below are taken using a code compiled with MDK-ARM toolchain V5.14 and with optimization Level3(-O3).

3.3 Measured current consumption

This section compares the current consumptions obtained in the following conditions:

- Flash memory stopped and CPU in Sleep mode
- Flash memory active and the CPU in Sleep mode.

Table 3 describes the average measures obtained for the MEMS loop part (SLEEP5, RUN7, SLEEP6 and RUN8) in the use case state machine.

Table 3. Current consumption example

Configuration Menu	Active peripherals	Current Consumption (uA)
Flash active while CPU in sleep	FLASH, RTC, USART2, DMA1(only stream0 and stream 6 enabled), I2C1	970
Stop Flash while CPU in sleep (BAM)	RTC, USART2, DMA1(only stream0 and stream 6 enabled), I2C1	710

Table 3 shows a significant current consumption reduction of around 27 % obtained with the new STM32F410/STM32F411/STM32F412/STM32F413 *Stop Flash memory while CPU in Sleep* feature (BAM).

4 Conclusion

This application note complements the datasheets and the reference manual by presenting a user guide on the new BAM implementation offered by the STM32F410, STM32F411, STM32F412 and STM32F413 lines.

A use case highlights the significant current consumption reduction obtained by using the BAM (Stop flash while CPU in sleep mode). It also allows assessing the different STM32F410/STM32F411/STM32F412/STM32F413 low-power modes and, thanks to the timestamp log to, determining all state periods and calculating the application average consumption.

5 Revision history

Table 4. Document revision history

Date	Revision	Changes
19-Dec-2014	1	Initial release.
08-Jun-2015	2	Removed table 1 Applicable products and software since the document describes a firmware (STSW-STM32154). Replaced LSM303DLHC by LSM6DS0 MEMS and 32F401CDISCOVERY by X-NUCLEO IKS01A1 in the whole document. Updated *Figure 2: Batch Acquisition Mode use case*, *Figure 6: MDK-ARM file placement*. Updated GPIO list and removed I2C1 in *Section 2.2: Peripherals used in STM32F410, STM32F411, STM32F412 and STM32F413 lines*. Updated use case step 2 in *Section 2.3: Functional description*. Updated *Section 3.1: Hardware requirements*. Updated MKD-ARM release version in *Section 3.2: Software settings*.
17-Aug-2015	3	Replaced STSW-STM32154 by X-CUBE-BAM.
22-Oct-2015	4	Added STM32F410xx part numbers and NUCLEO-F410RB. Updated title.
04-Jan-2016	5	Added STM32F412xx part numbers and NUCLEO-F412ZG.
06-Jan-2016	6	Minor modifications in *Section 2.3: Functional description* and *Section 3.1: Hardware requirements*.
21-Nov-2016	7	Updated the whole document adding the STM32F413 line. Added *Section 1.3.6: How to execute code from RAM using AC6-SW4STM32 toolchain*.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2016 STMicroelectronics – All rights reserved