STM32 AZURE RTOS Workshop

FileX

Yilmaz KASAPOGLU

Agenda

1 FileX/LevelX Overview

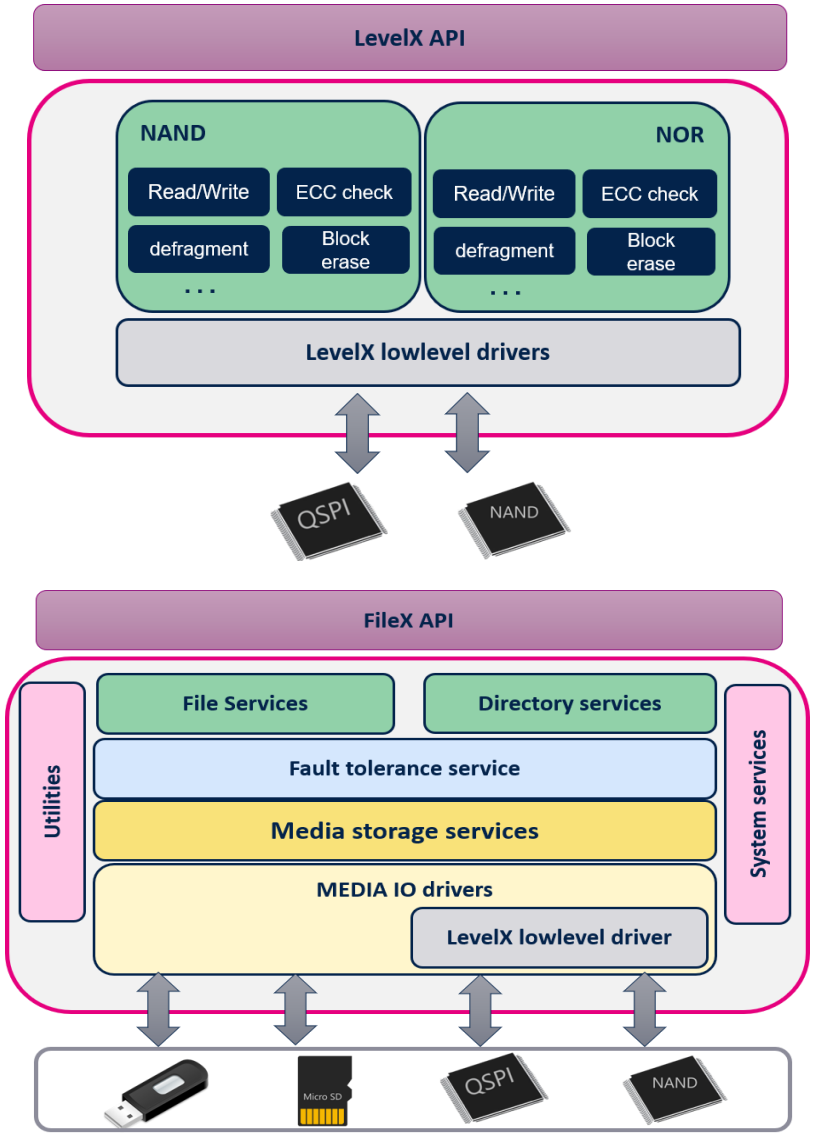
2 STM32 Integration with STM32CubeMX

3 Lab

FileX/LevelX Overview: features

- FileX is a high-performance, file allocation table (FAT)-compatible file system.
  - Storage media management
  - Multi partition support
  - File/Directory Access
  - Fault tolerance
  - Support all common media storage devices
- LevelX is a library that provides API to deal with NAND and NOR flash memories
  - Read/Write Access
  - Wear-leveling to increase disk life time
  - Defragmentation
  - Bad block management

Currently FileX and LevelX requires ThreadX, but there is a plan from MSFT to provide a baremetal version.



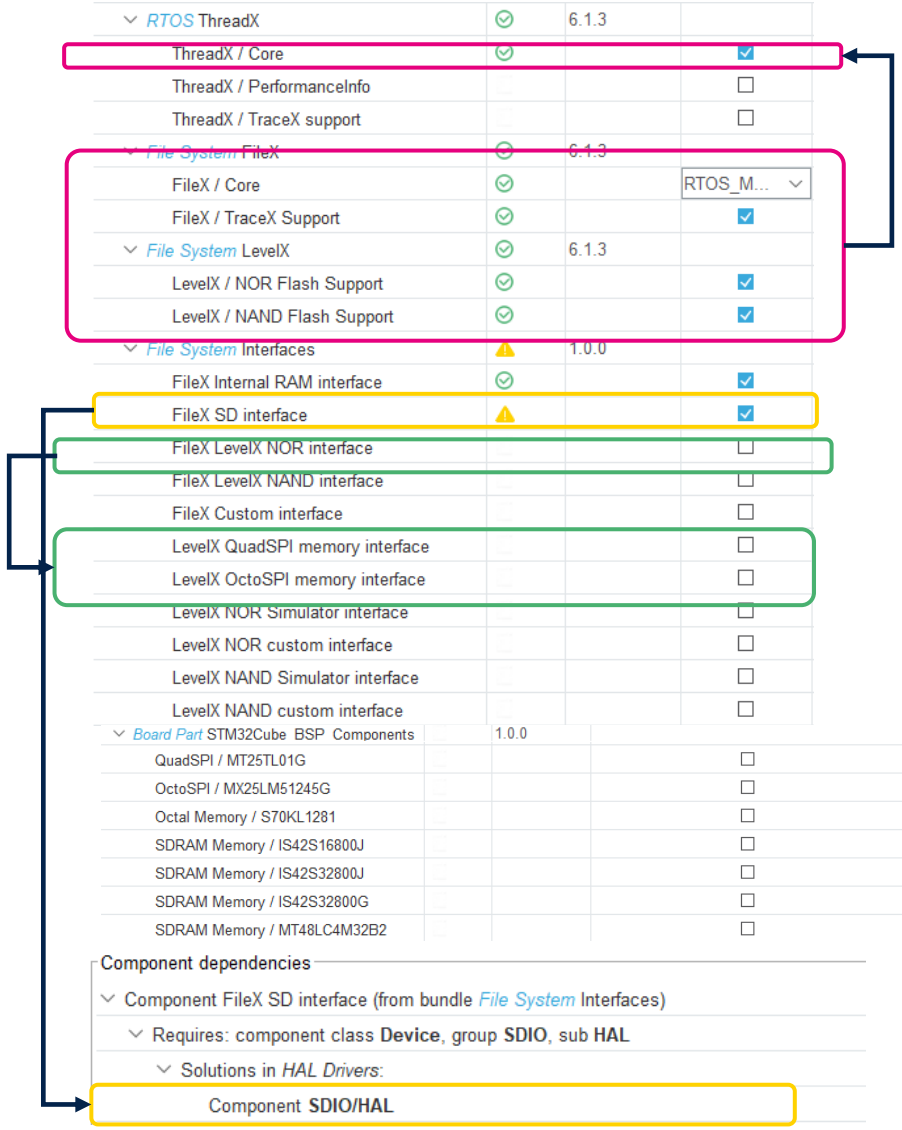
FileX/LevelX Overview: APIs

FileX Feature	API	Comments
FileSystem	fx_system_xxx()	Internal init, system time get/set.
Media device	fx_media_xxx()	Managing the media device: format, open, close…
File management	fx_file_xxx() fx_unicode_file_xxx()	Handling file: create, read, write, seek, delete…
Directory management	fx_directory_xxx()	Handling directories:create, delete, find…

LevelX Feature	API	Comments
NAND Flash support	lx_nand_flash_xxx()	NAND flash management: open, close, read block,…
NOR Flash support	lx_nor_flash_xxx()	NOR flash management: open, close, erase block,…

STM32CubeMX: component sector UI

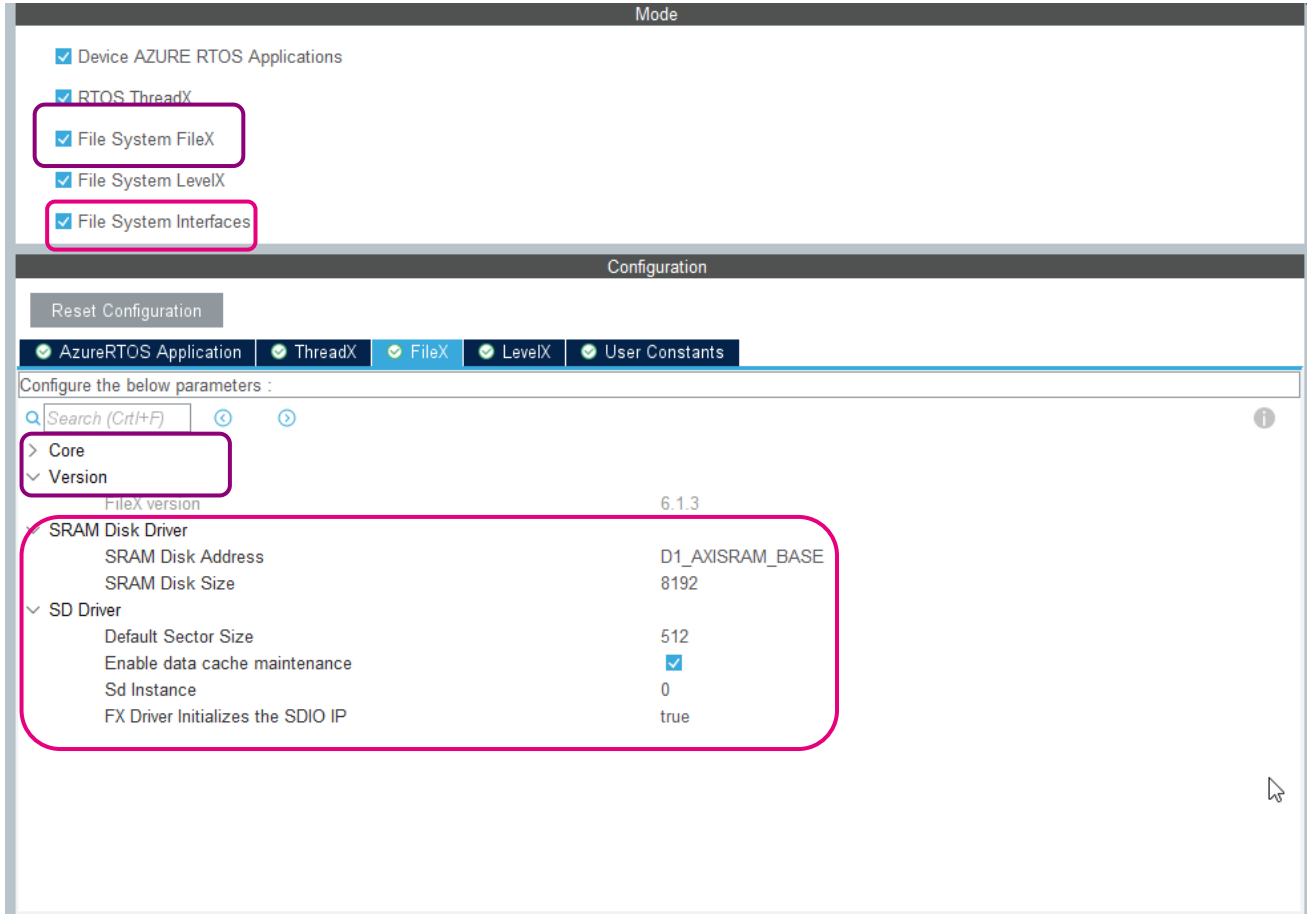
- FileX/LevelX current version requires ThreadX.
- Low level interfaces for FileX and LevelX are exposed in CubeMX. Some of them are using the BSP/Drivers to be enhanced in the coming updates.
- It is possible to instantiate different interfaces (SD, OSPI,…)
- Some Interfaces require that the IP HW is enabled in STM32CubeMX.
- Instantiating the same interface multiple times is supported by FileX but not possible from STM32CubeMX UI
- FileX USB MSC is managed by the class USBX Host Storage class.



life.augmented

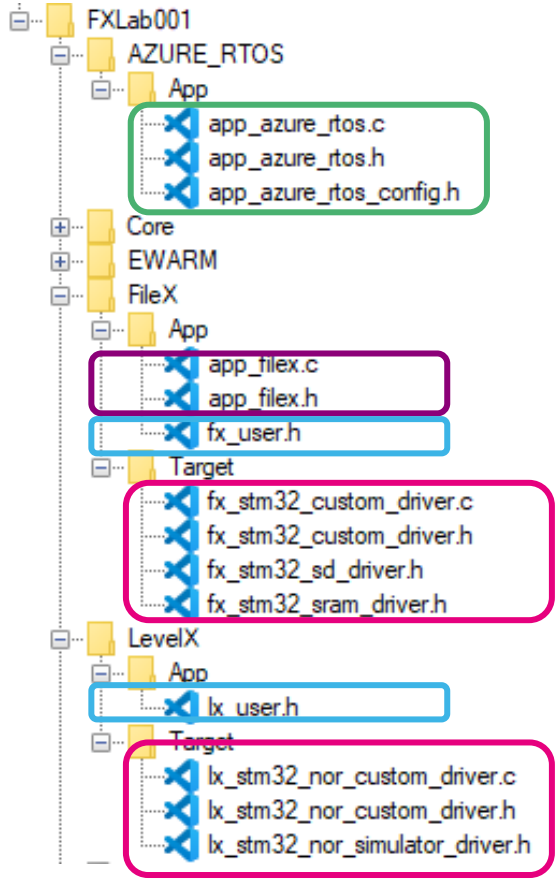
STM32CubeMX: Config UI

- STM32CubeMX Config UI allows the users to:
  - configure the FileX/LevelX core flags.
  - The config flags are written in the file “fx_user.h” generated by STM32CubeMX.
  - Configure specific options for the low-level Interfaces



STM32CubeMX: Project generation content

- app_azure_rtos.c/.h app_azure_rtos_config.h: files for application start (MX_AZURE_RTOS_Init())
- app_filex.c/.h :user application code files to implement FileX support
- fx_user.h/lx_user.h: FileX/LevelX config files.
- fx_stm32_xxx_driver.h/c : driver files for the low-level interfaces for FileX
- When a custom driver is needed, the driver files should be implemented under the application.

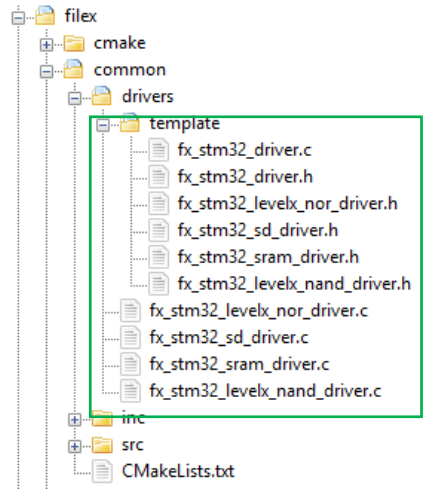


life.augmented

CubeAzure FileX: LowLevel Drivers

STM32 FileX HW low-level drivers:

- FileX drivers allow the interaction with physical media storage
- When combined with LevelX it is possible to have a FileSytem on NOR and NAND memories.
- Each driver is split in 2 files:
  - Driver implementation provided as “pattern”
  - Driver header provided as template at application level.



FileX lowLevel drivers

Link to FileX low-level drivers

- Each FileX driver has a main entry function that is registered by the main application like via functions “fx_media_format()” or “fx_media_open()” functions

```
VOID  fx_stm32_sd_driver(FX_MEDIA *media_ptr)
{
    UINT status;
    UINT unaligned_buffer = 0;
    ULONG partition_start;
    ULONG partition_size;

#if (FX_DRIVER_CALLS_SD_INIT == 0)
    is_initialized = 1; /* the SD  was initialized by the application*/
#endif
   /* before performing any operation, check the status of the SDMMC */
    if (is_initialized == 1)
    {
        if (check_sd_status(SD_INSTANCE) != BSP_ERROR_NONE)
        {
            media_ptr->fx_media_driver_status =  FX_IO_ERROR;
            return;
        }
    }

    /* Process the driver request specified in the media control block.  */
    switch(media_ptr->fx_media_driver_request)
    {
        case FX_DRIVER_INIT:
        {
#if (FX_DRIVER_CALLS_SD_INIT == 1)
            /* Initialize the SD instance */
            if (is_initialized == 0)
            {
                status = fx_sd_driver_init(SD_INSTANCE);

                if (status == BSP_ERROR_NONE)
                {
                    is_initialized = 1;
#endif
                    /* Create a counting semaphore to check the DMA transfer status */
                    if (tx_semaphore_create(&transfer_semaphore, "sdmmc dma transfer semaphore", 1) != TX_SUCCESS)
                    {
                        media_ptr->fx_media_driver_status =  FX_IO_ERROR;
                    }
                    else
                    {
                        media_ptr->fx_media_driver_status =  FX_SUCCESS;
                    }
#if (FX_DRIVER_CALLS_SD_INIT == 1)
                }
                else
                {
                    media_ptr->fx_media_driver_status =  FX_IO_ERROR;
                }
            }
#endif
            break;
        }

        case FX_DRIVER_UNINIT:
```

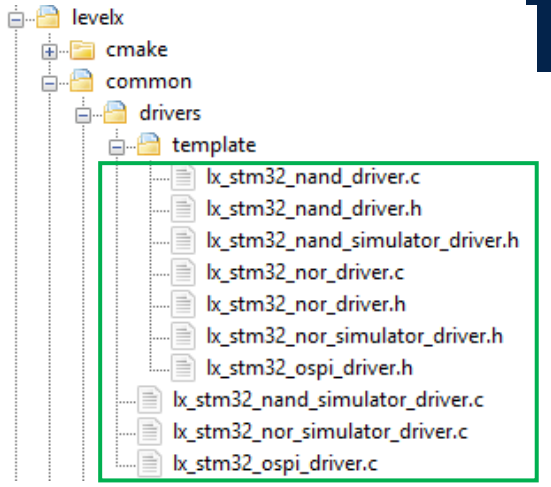
```
/* Open the SD disk driver.  */
status =  fx_media_open(&sdio_disk, "STM32_SDIO_DISK", fx_stm32_sd_driver, 0,(VOID *) media_memory, sizeof(media_memory));
```

life.augmented

CubeAzure LevelX: LowLevel Drivers

Link to LevelX low-level drivers

- Both NOR and NAND drivers requires a main entry function that is the unique interface to handle the LevelX core stack requests (read, write, erase block…)
- The driver is registered by the application when needed.



LevelX low-level drivers.

```
#include "lx_api.h"

UINT  lx_stm32_qspi_initialize(LX_NOR_FLASH *nor_flash);
```

```
UINT  lx_nor_flash_open(LX_NOR_FLASH  *nor_flash, CHAR *name, UINT (*nor_driver_initialize)(LX_NOR_FLASH *));
```

```
UINT lx_stm32_qspi_initialize(LX_NOR_FLASH *nor_flash)
{
 BSP_QSPI_Info_t qspi_info;

 if (is_initialized == LX_FALSE)
 {
   if(BSP_QSPI_GetInfo(QSPI_INSTANCE, &qspi_info) != BSP_ERROR_NONE)
   {
     return LX_ERROR;
   }

#if (LX_DRIVER_CALLS_QSPI_INIT == 1)

   TX_INTERRUPT_SAVE_AREA
     BSP_QSPI_Init_t qspi_config;

   /* QSPI device configuration */
   qspi_config.InterfaceMode = BSP_QSPI_QPI_MODE;
   qspi_config.TransferRate  = BSP_QSPI_DTR_TRANSFER;

   TX_DISABLE

     if(BSP_QSPI_Init(QSPI_INSTANCE, &qspi_config) != BSP_ERROR_NONE)
     {
       return LX_ERROR;
     }

#if (LX_DRIVER_ERASES_QSPI_AFTER_INIT == 1)
   if( BSP_QSPI_EraseChip(QSPI_INSTANCE) != BSP_ERROR_NONE)
   {
     return LX_ERROR;
   }

   if(check_status() != LX_SUCCESS)
   {
     return LX_ERROR;
   }

#endif

   TX_RESTORE

#endif
```

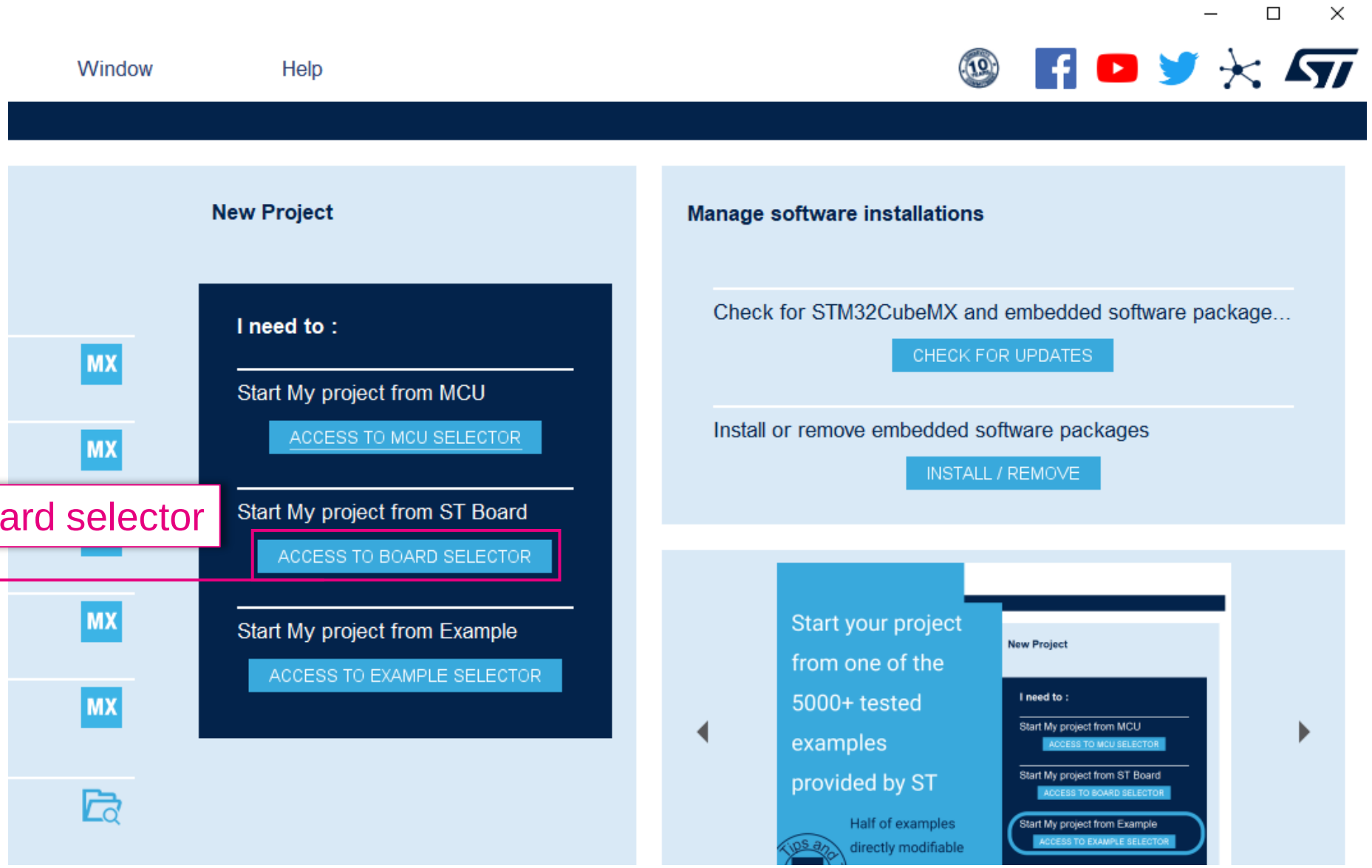
Lab – FileX Implementation

Purpose

- Implementing FileX support using STM32CubeMx from scratch

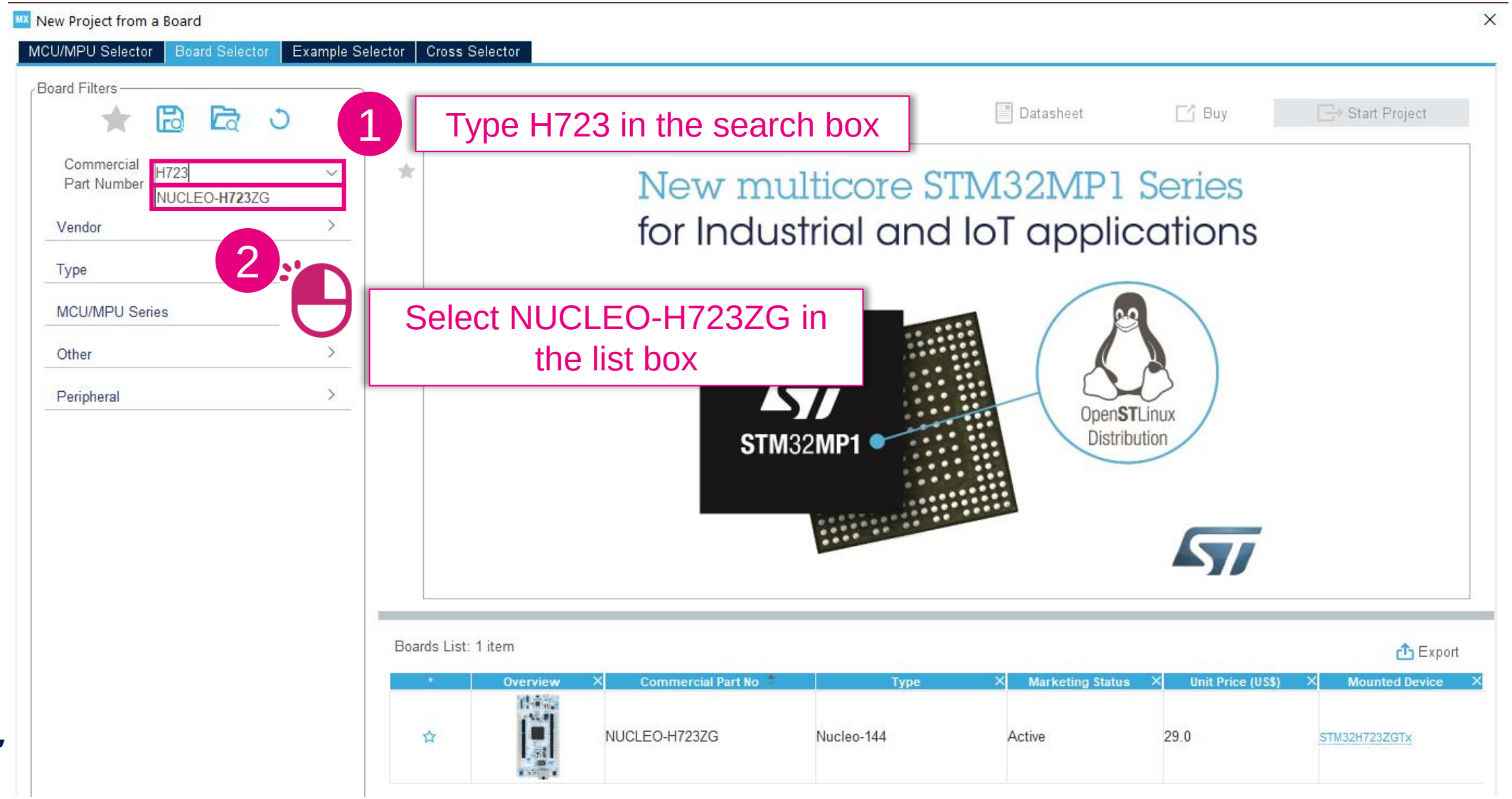
- By this lab, we will;
  - create a file system on the internal SRAM,
  - open the file in write mode and write data,
  - re-open the file in read mode and read the file content.

Launch STM32CubeMx

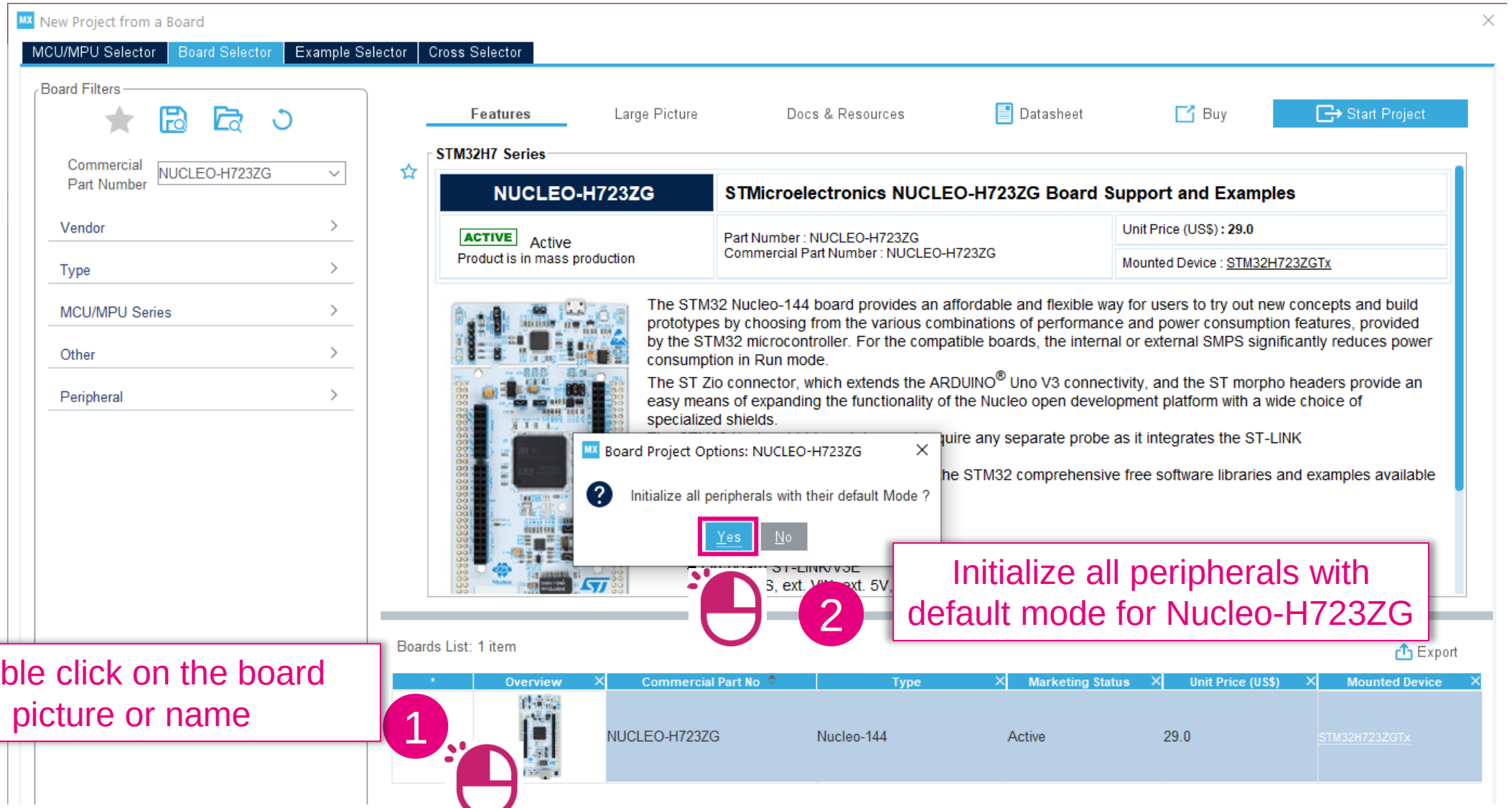


Start new project with board selector

STM32CubeMX Board Selector



STM32CubeMX Board Selector

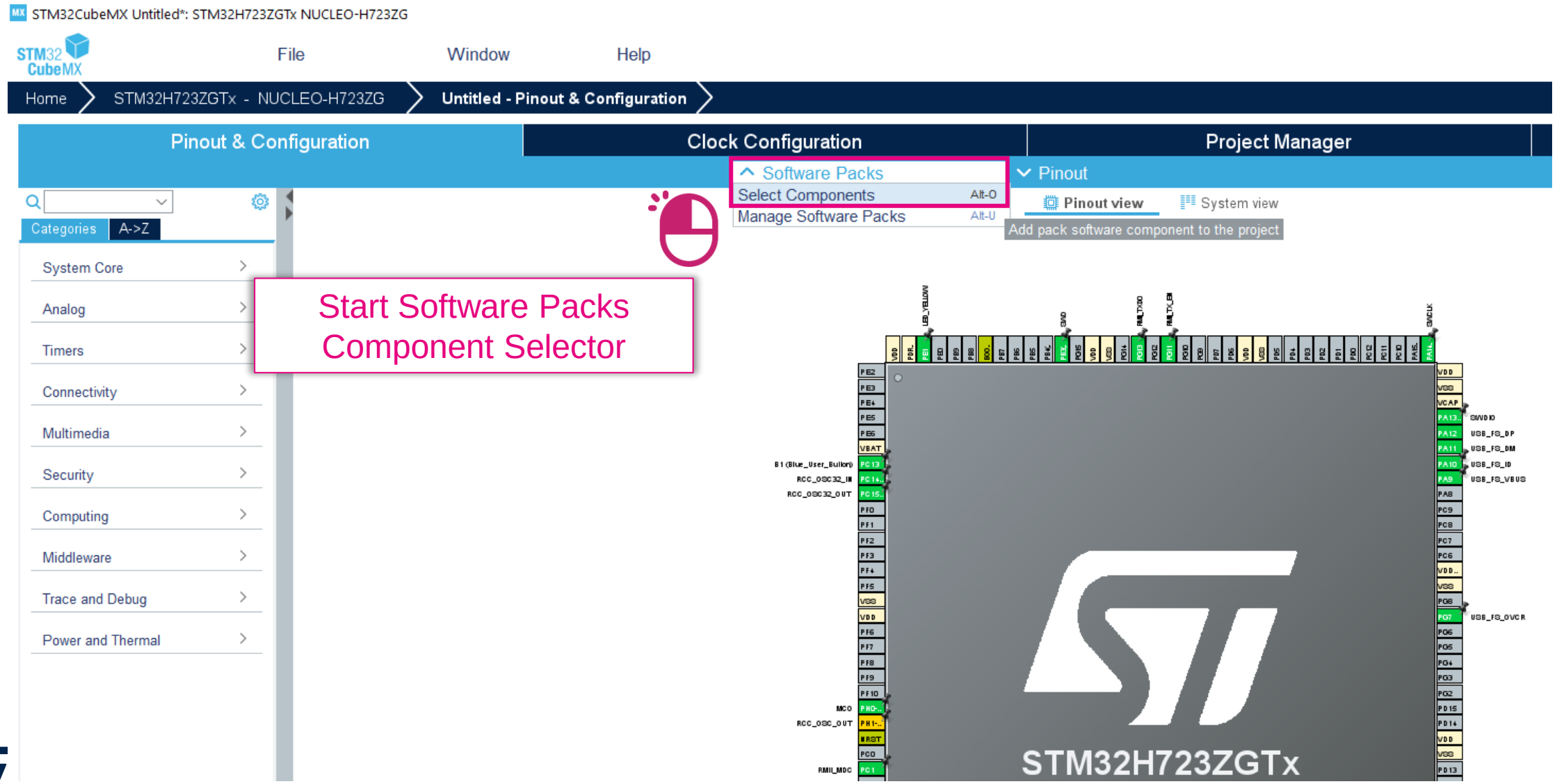


Double click on the board picture or name

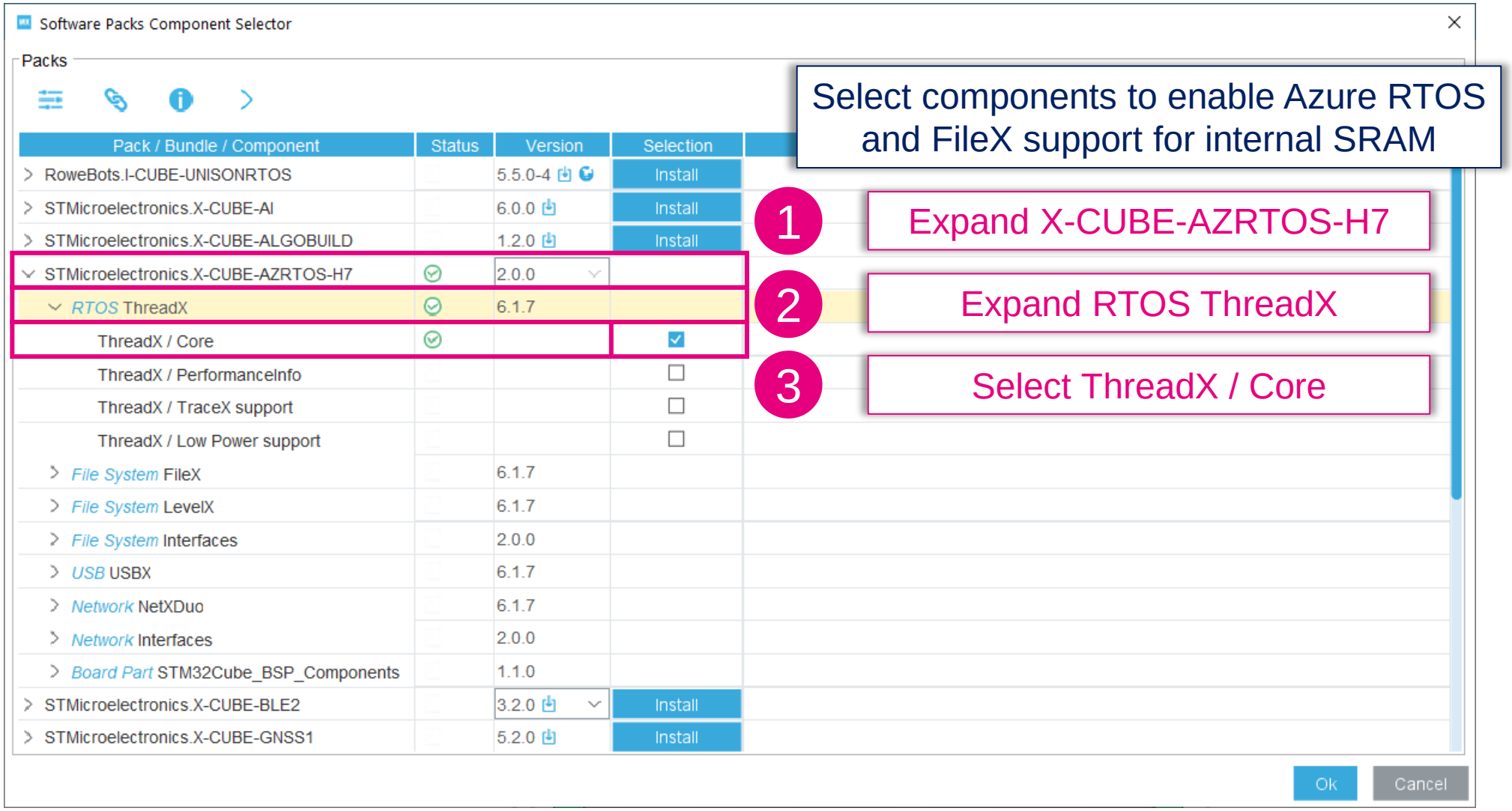
Initialize all peripherals with default mode for Nucleo-H723ZG

life.augmented

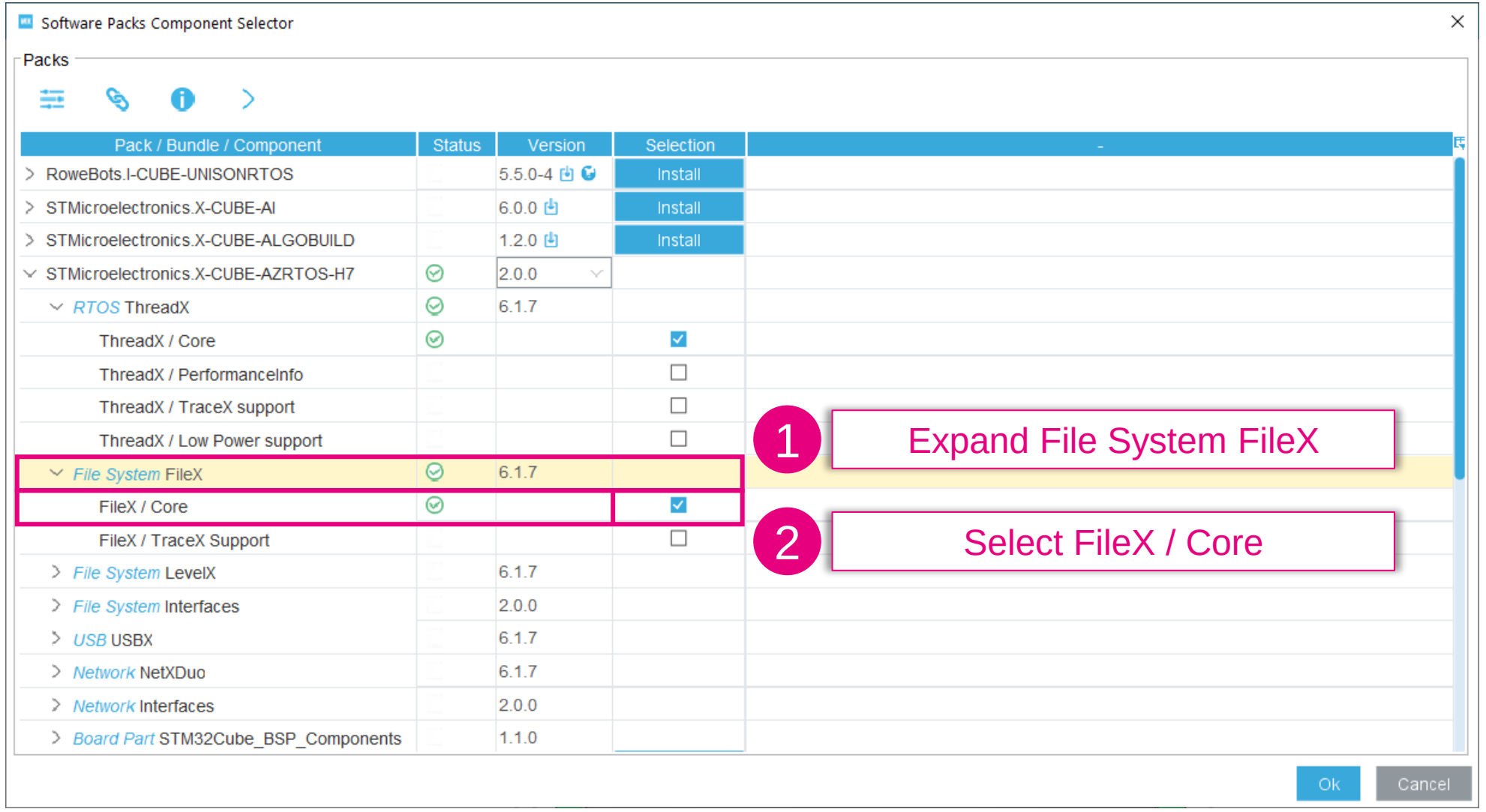
Add X-CUBE-AZRTOS-H7 Support



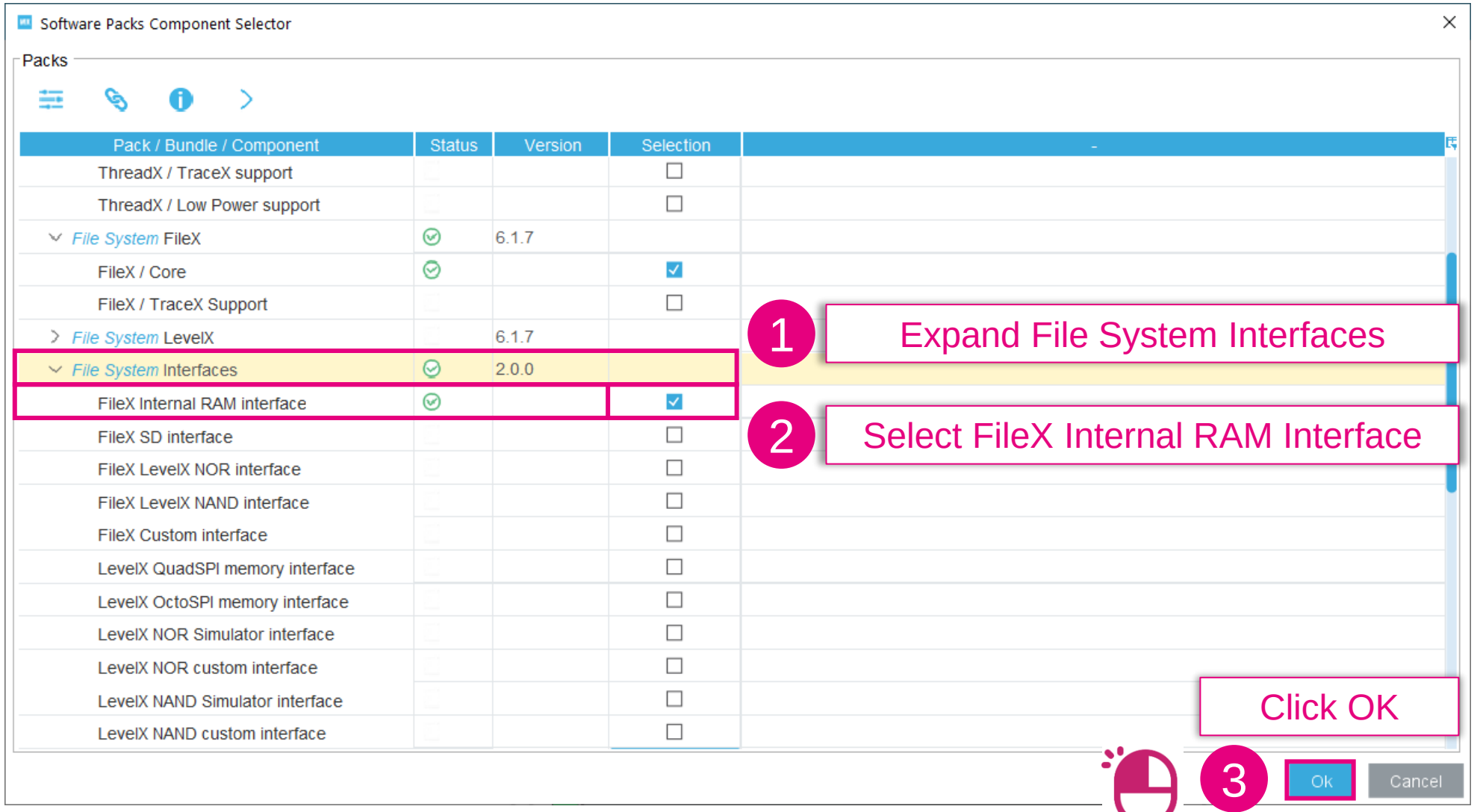
Add X-CUBE-AZRTOS-H7 Support



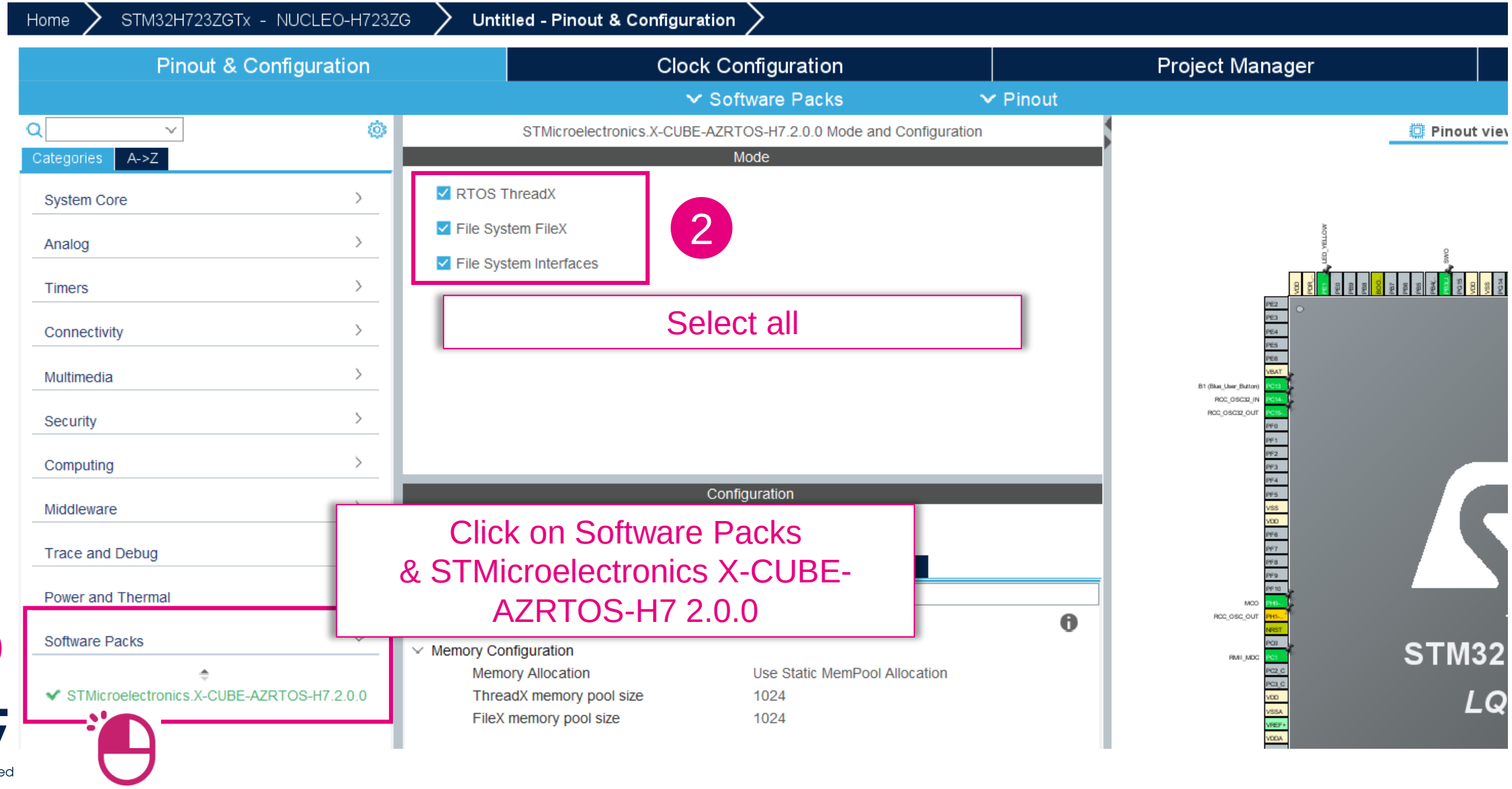
Add X-CUBE-AZRTOS-H7 Support



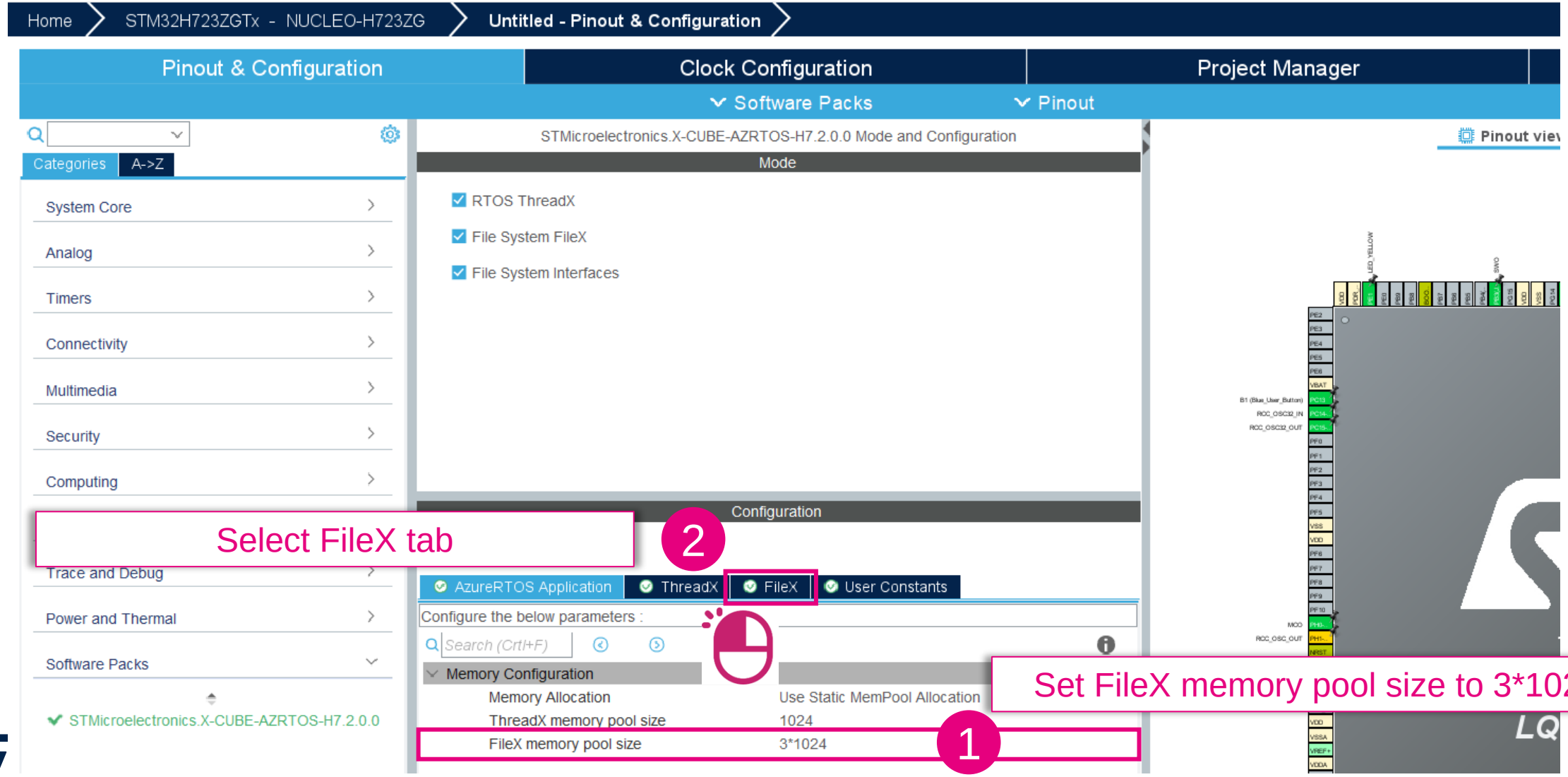
Add X-CUBE-AZRTOS-H7 Support



Configure X-CUBE-AZRTOS



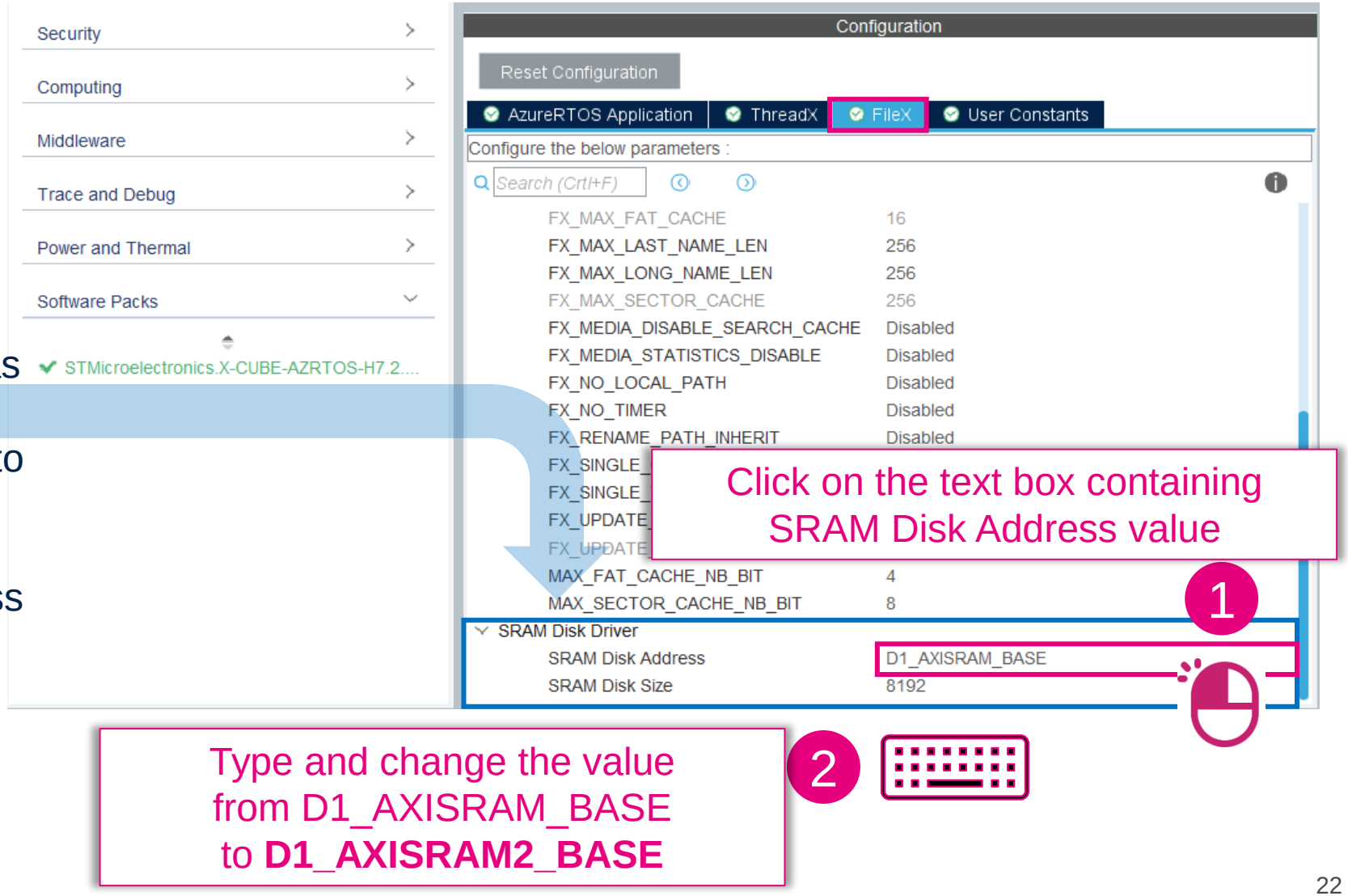
Configure X-CUBE-AZRTOS



SRAM Disk Driver

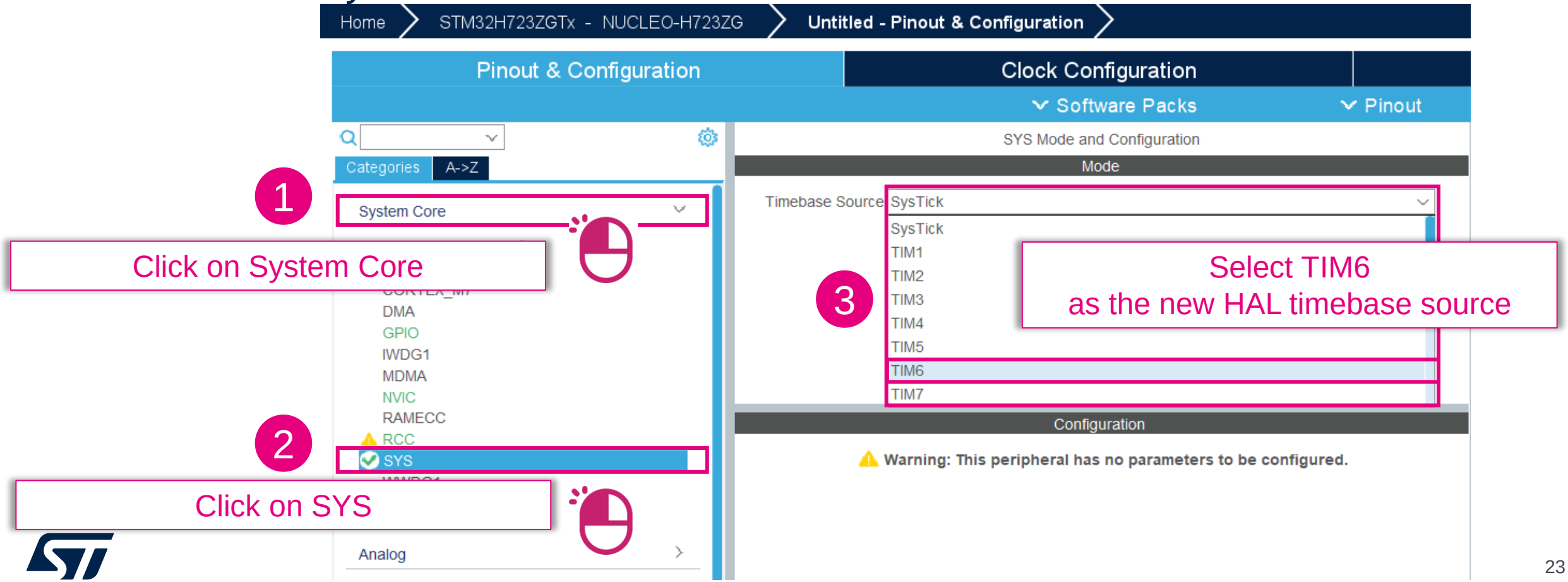
Please note that;

- **SRAM Disk Address** is configured as **D1_AXISRAM_BASE** by CubeMX, however this address space is used to store FileX version information.
- Therefore we will use another address space in the internal RAM to create our disk media.

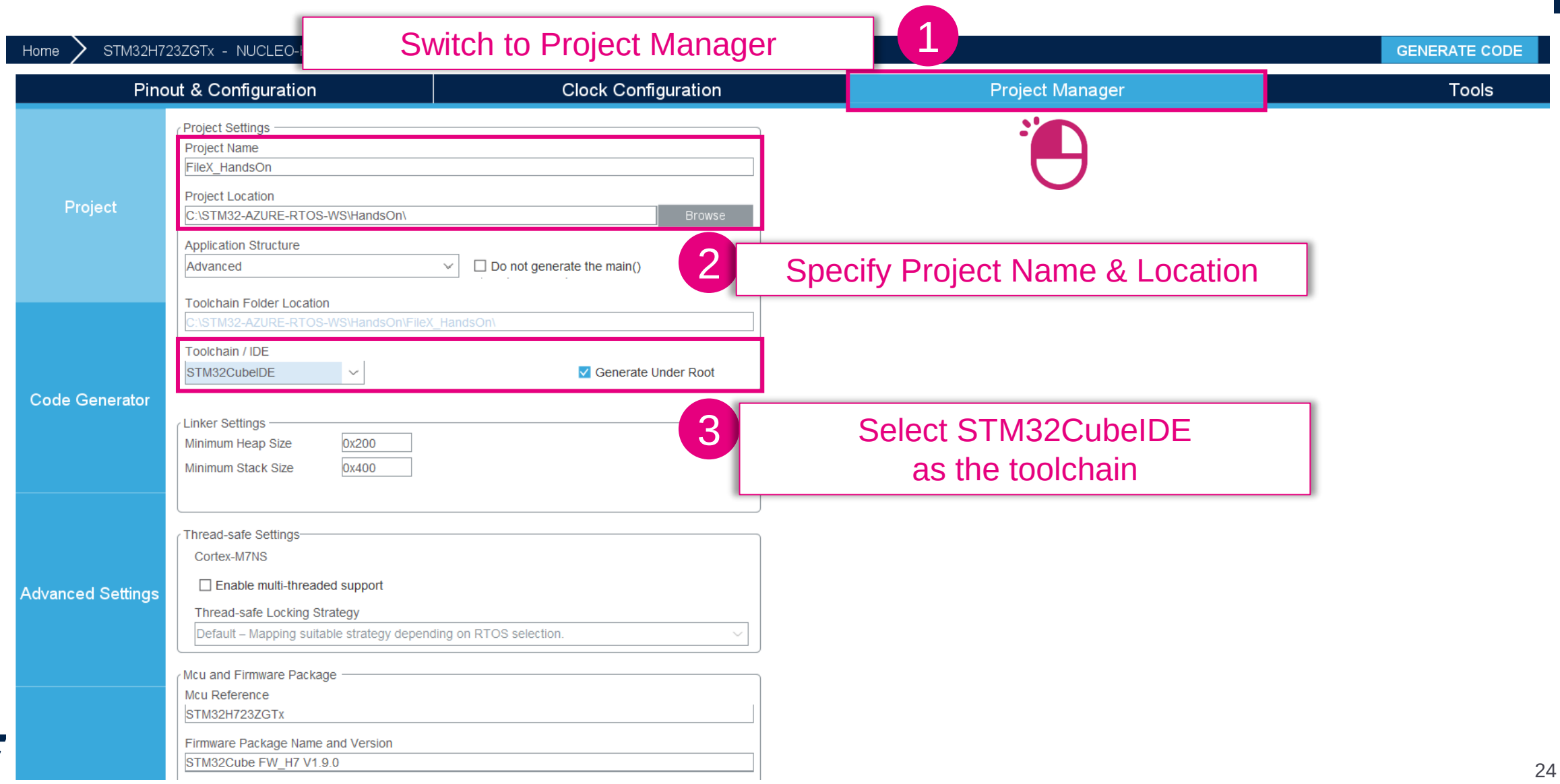


Set Timebase Source for the RTOS

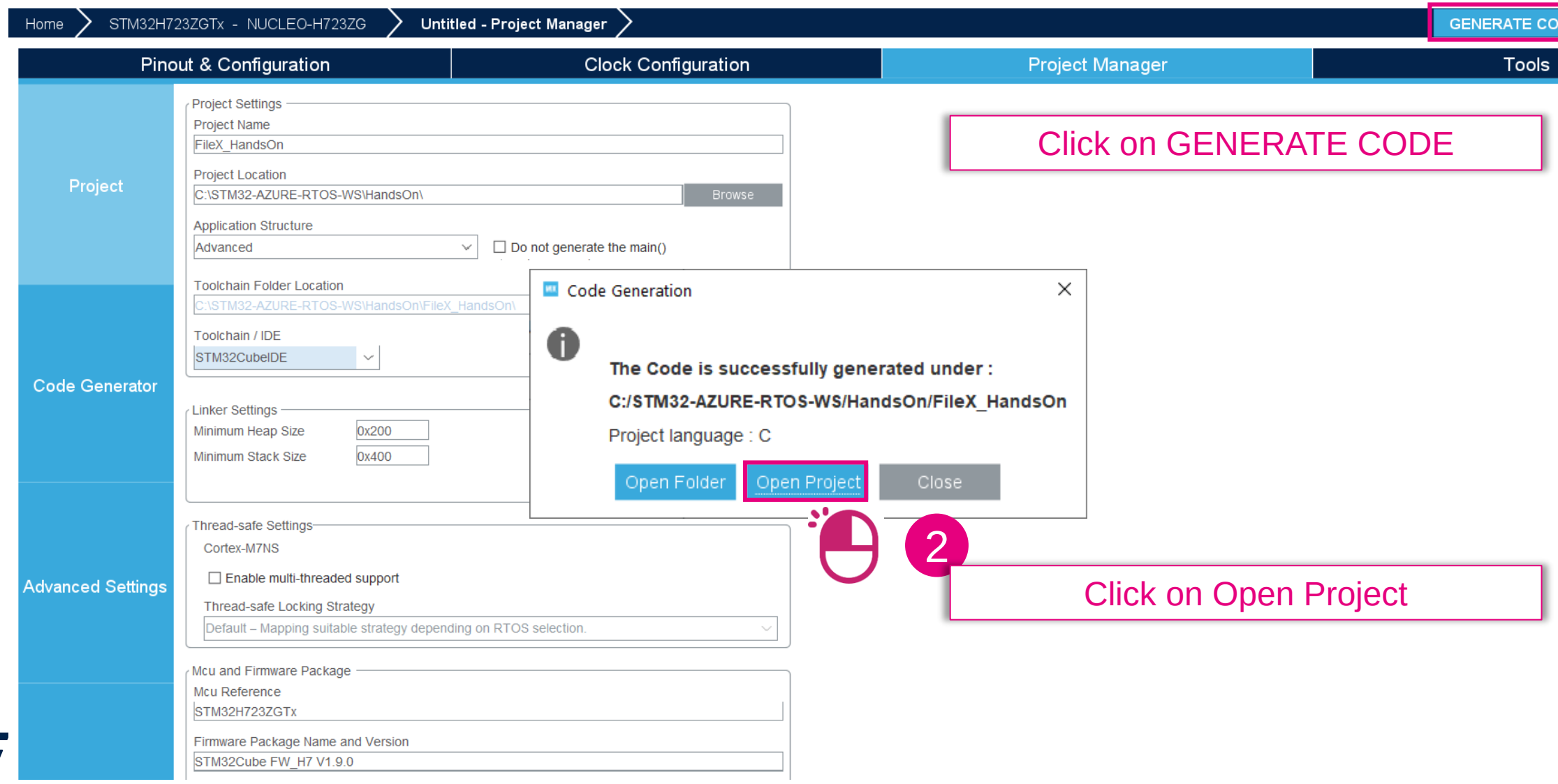
- When RTOS is used, it is strongly recommended to use a HAL timebase source other than the Systick.



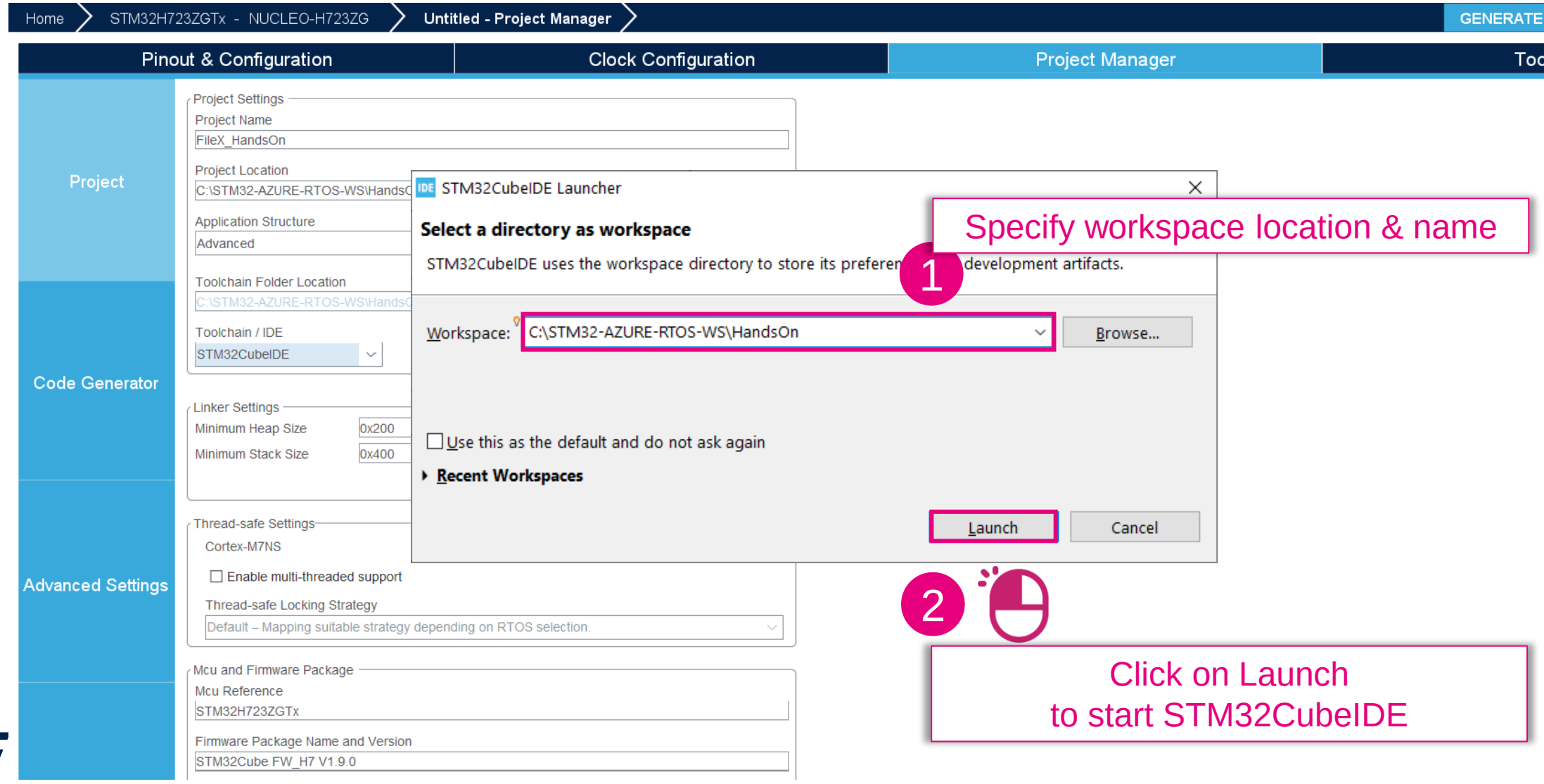
Project Settings



Code Generation

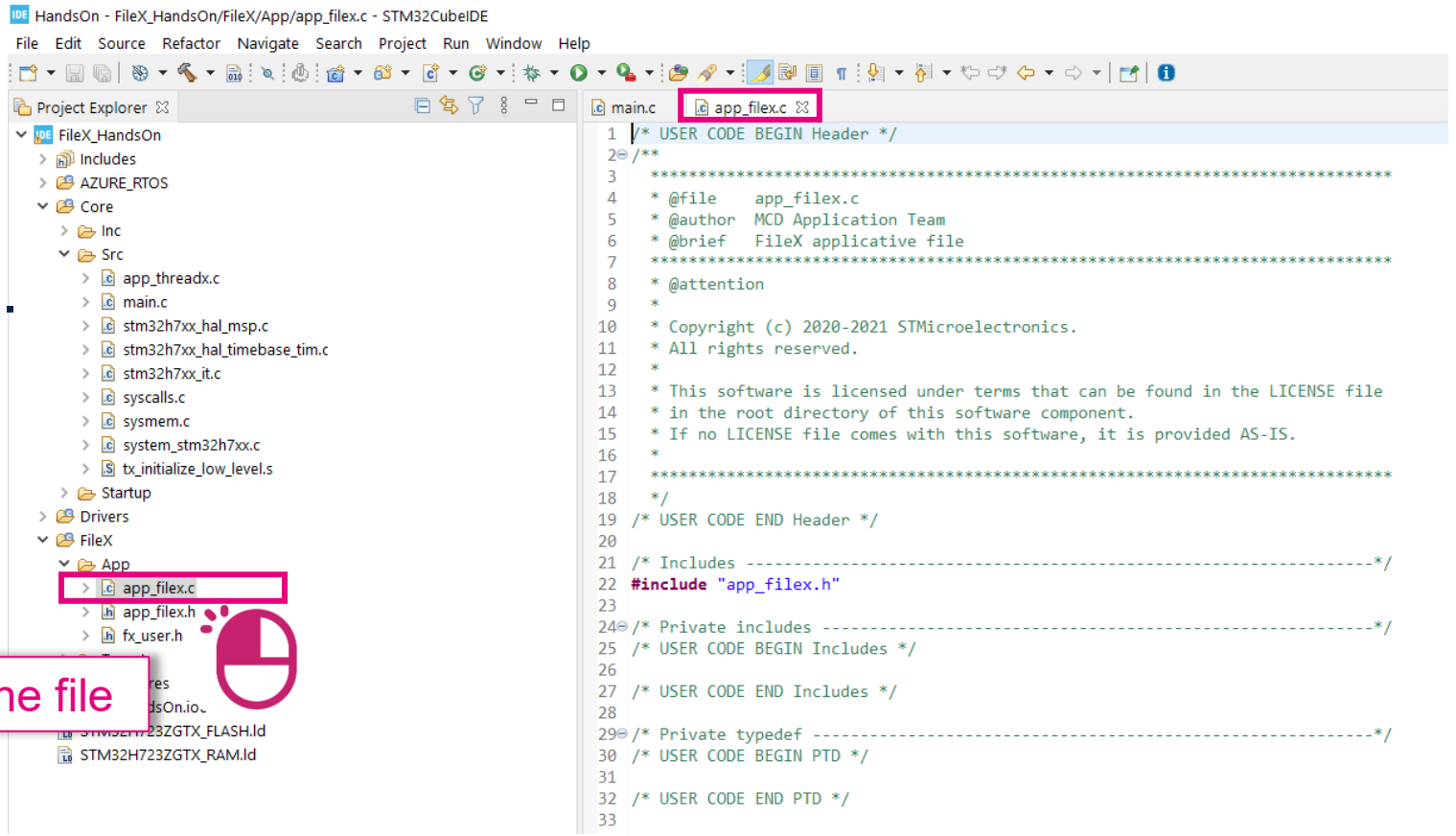


Launch STM32CubeIDE



Generated Code

- Generated code is a buildable project with the project tree structure seen on the right.
- Azure RTOS and FileX support added to the project by CubeMX.
- We will need to **modify “app_filex.c”** to implement our file system.

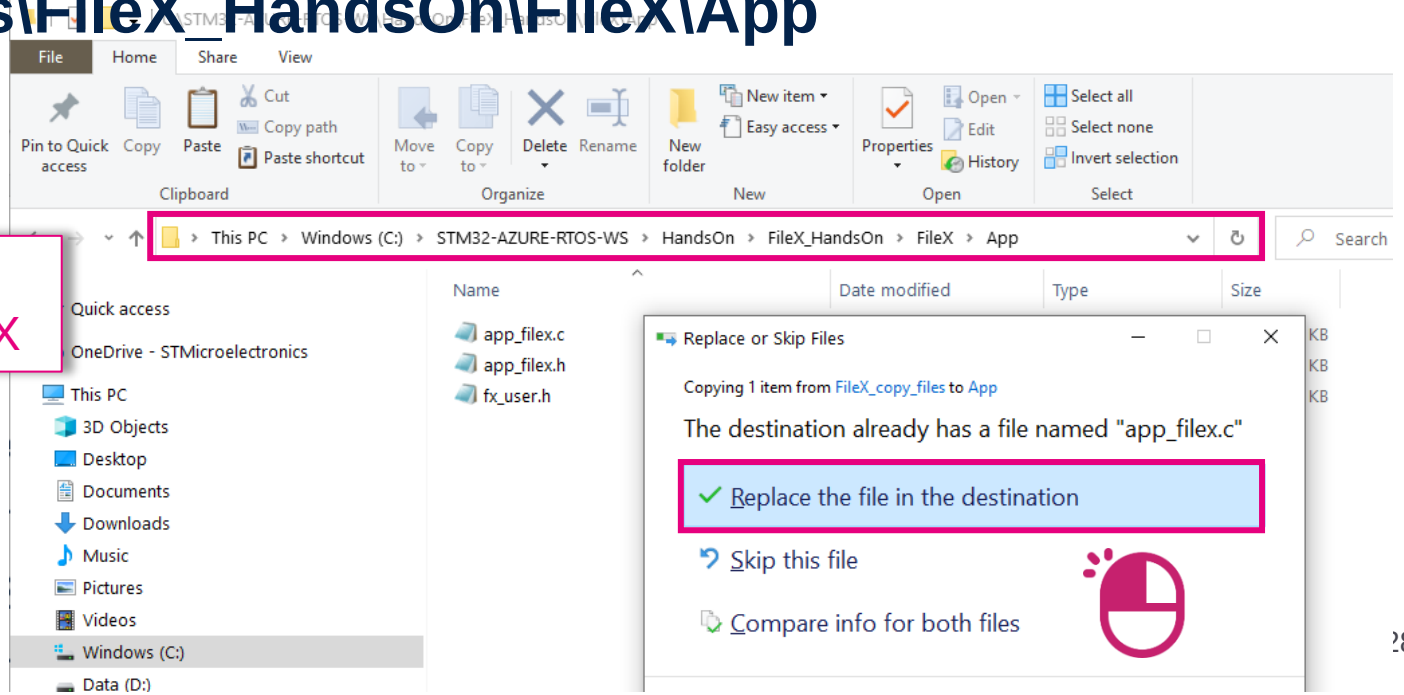


Double click on “app_filex.c” to edit the file

Implementing the File System

- At this step we will use completed code to save time
- Path for the completed code :
  C:\STM32-AZURE-RTOS-WS\HandsOn\FileX_copy_files
- Path for the FileX application files in the generated project :
  C:\STM32_AzureRTOS_WS\Labs\FileX_HandsOn\FileX\App

Copy the completed **“app_filex.c”** file & replace it with the one generated by CubeMX



Implementing the File System

- In the completed **“app_filex.c”** file, defined by the user:
- **Default sector and stack sizes,**
- **Default thread priority**
- **Default preemption threshold**

- **fX_thread_entry ()** is the main thread and should be implemented by the user after the code generation

main.c | app_filex.c

```
34 /* Private define ------------------------------------------------------------*/
35 /* USER CODE BEGIN PD */
36 #define DEFAULT_SECTOR_SIZE             512
37 #define DEFAULT_STACK_SIZE              (2 * 1024)
38 /* Thread_0 priority */
39 #define DEFAULT_THREAD_PRIO             10
40 /* Thread_0 preemption priority */
41 #define DEFAULT_PREEMPTION_THRESHOLD     DEFAULT_THREAD_PRIO
42 /* USER CODE END PD */
43
44 /* Private macro -------------------------------------------------------------*/
45 /* USER CODE BEGIN PM */
46
47 /* USER CODE END PM */
48
49 /* Private variables ---------------------------------------------------------*/
50 /* USER CODE BEGIN PV */
51 UCHAR            *media_memory;
52 /* Define FileX global data structures.  */
53 FX_MEDIA         ram_disk;
54 FX_FILE          fx_file;
55 /* Define ThreadX global data structures.  */
56 TX_THREAD        fx_thread;
57 /* USER CODE END PV */
58
59 /* Private function prototypes -----------------------------------------------*/
60 /* USER CODE BEGIN PFP */
61 void fx_thread_entry(ULONG thread_input);
62 void Error_Handler(void);
63 /* USER CODE END PFP */
64
```

FileX thread entry function defined by the user

life.augmented

Implementing the File System

- App_FileX_Init() is the only function generated by CubeMx in “app_filex.c” file

```
70 UINT MX_FileX_Init(VOID *memory_ptr)
71 {
72   UINT ret = FX_SUCCESS;
73   TX_BYTE_POOL *byte_pool = (TX_BYTE_POOL*)memory_ptr;
74
75   /* USER CODE BEGIN MX_FileX_MEM_POOL */
76   (void)byte_pool;
77   /* USER CODE END MX_FileX_MEM_POOL */
78
79   /* USER CODE BEGIN MX_FileX_Init */
80   VOID *pointer;
81
82   /* Allocate memory for the FileX thread's stack */
83   ret = tx_byte_allocate(byte_pool, &pointer, DEFAULT_STACK_SIZE, TX_NO_WAIT);
84
85   if (ret != FX_SUCCESS)
86   {
87     /* Failed at allocating memory */
88     Error_Handler();
89   }
90
```

We need to allocate the memory for the stack of FileX thread with the DEFAULT_STACK_SIZE defined by us on top of the file

life.augmented

Implementing the File System

```
88      Error_Handler();
89    }
90
91    /* Create the main thread.  */
92    tx_thread_create(&fx_thread, "thread 0", fx_thread_entry, 0, pointer, DEFAULT_STACK_SIZE, DEFAULT_THREAD_PRIO,
93            DEFAULT_THREAD_PRIO, TX_NO_TIME_SLICE, TX_AUTO_START);
94
95    /* Allocate memory for the media cache */
96    ret = tx_byte_allocate(byte_pool, (VOID**) &media_memory, DEFAULT_SECTOR_SIZE, TX_NO_WAIT);
97
98    if (ret != FX_SUCCESS)
99    {
100     /* Failed at allocating memory */
101     Error_Handler();
102   }
103
104   /* Initialize FileX.  */
105   fx_system_initialize();
106   /* USER CODE END MX_FileX_Init */
107   return ret;
108 }
109
110 /* USER CODE BEGIN 1 */
```

Main thread is created here with the function name **fx_thread_entry()**

Cache memory allocation for the disk media with the DEFAULT_SECTOR_SIZE

Finally, Azure RTOS FileX system is initialized

life.augmented

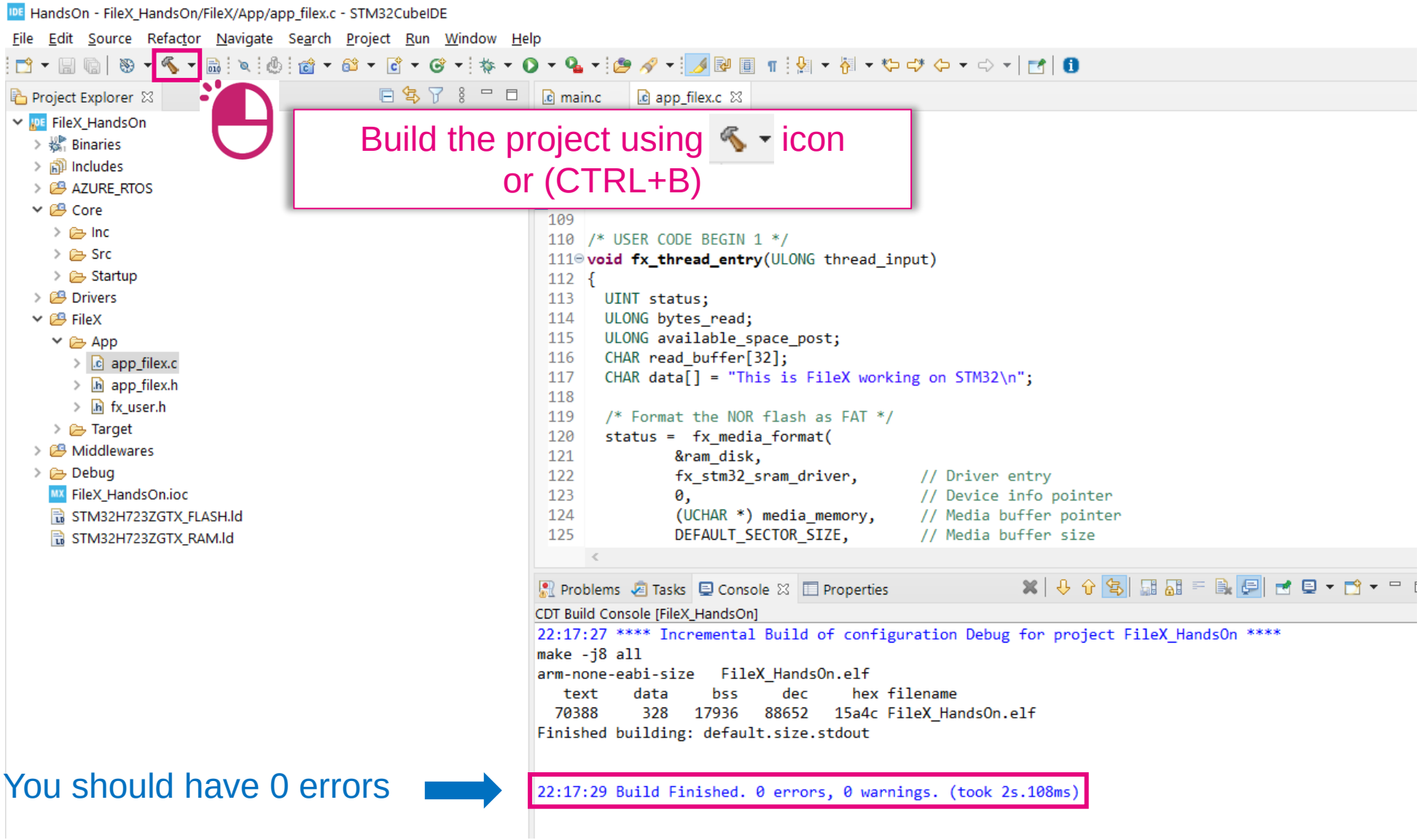
Through the Code

```
110 /* USER CODE BEGIN 1 */
111 void fx_thread_entry(ULONG thread_input)
112 {
113   UINT status;
114   ULONG bytes_read;
115   ULONG available_space_post;
116   CHAR read_buffer[32];
117   CHAR data[] = "This is FileX working on STM32\n";
118
119   /* Format the NOR flash as FAT */
120   status =  fx_media_format(
121           &ram_disk,
122           fx_stm32_sram_driver,       // Driver entry
123           0,                          // Device info pointer
124           (UCHAR *) media_memory,     // Media buffer pointer
125           DEFAULT_SECTOR_SIZE,        // Media buffer size
126           "RAM_DISK",                 // Volume Name
127           1,                          // Number of FATs
128           32,                         // Directory Entries
129           0,                          // Hidden sectors
130           64,                         // Total sectors
131           DEFAULT_SECTOR_SIZE,        // Sector size
132           8,                          // Sectors per cluster
133           1,                          // Heads
134           1                           // Sectors per track
135   );
136
```

Main thread function.
This part of the code will be explained during the debug process.

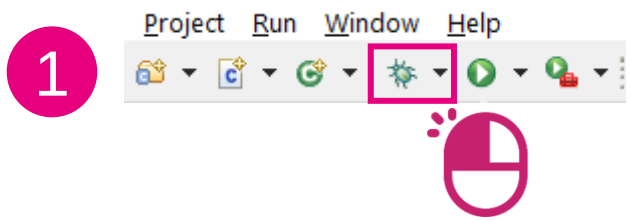
- Now we are ready to build the code

Build the Project



You should have 0 errors

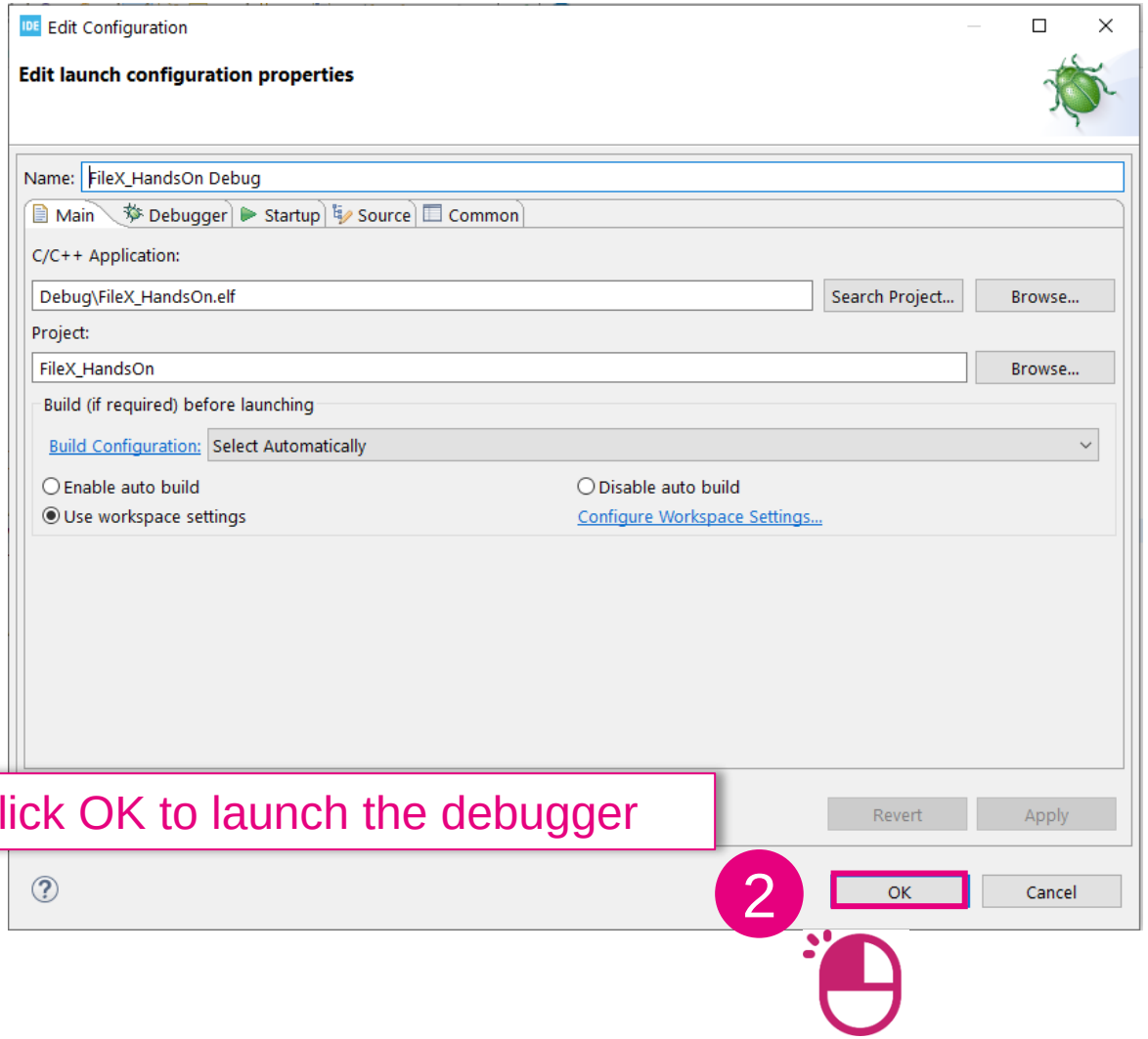
Start the Debugger



Click on debug icon
or (F11) to start the debugger

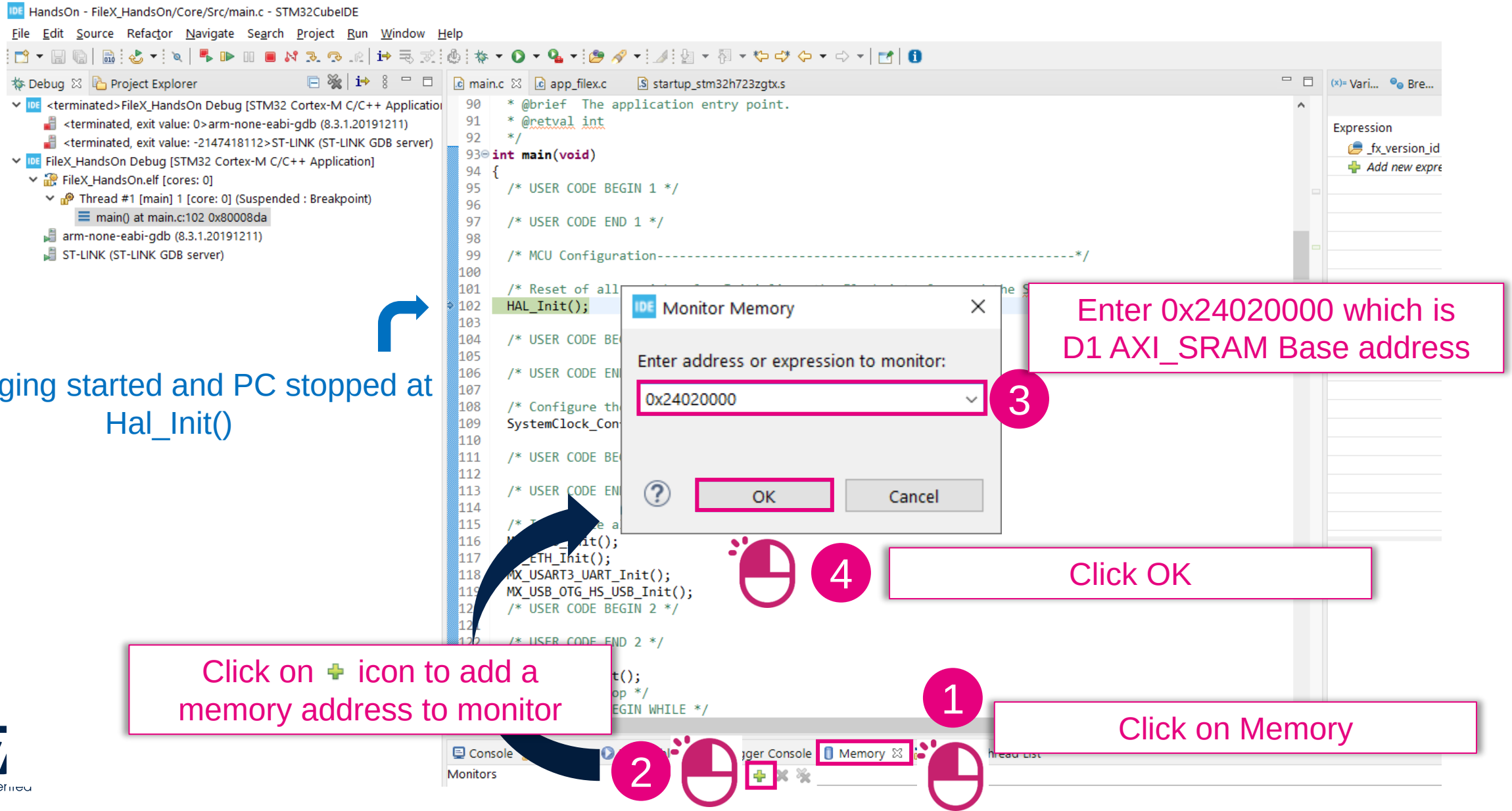
Debug Configuration Window will pop-up

- Default settings are OK for STLink



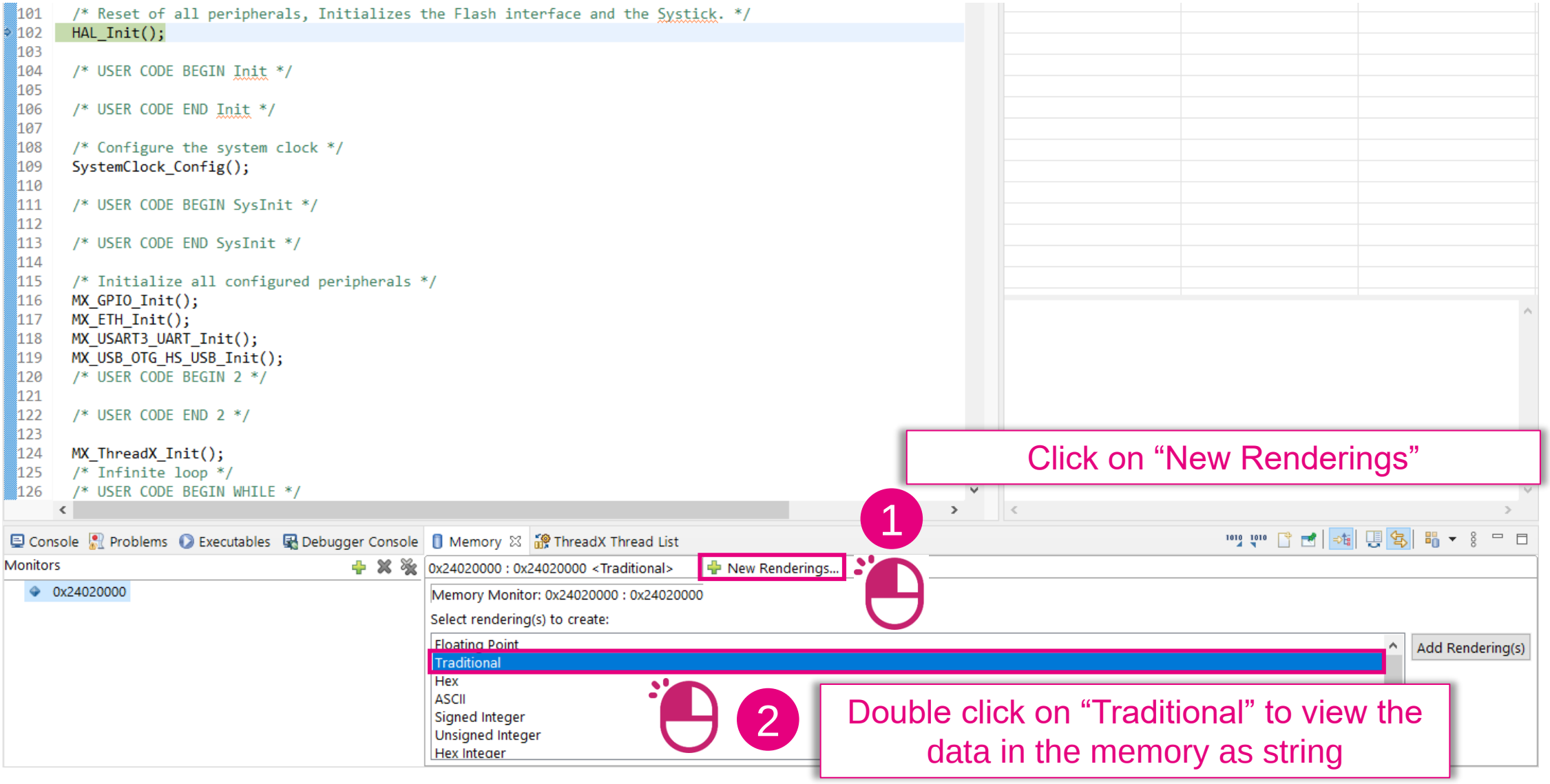
Click OK to launch the debugger

Debugging the FileX Code

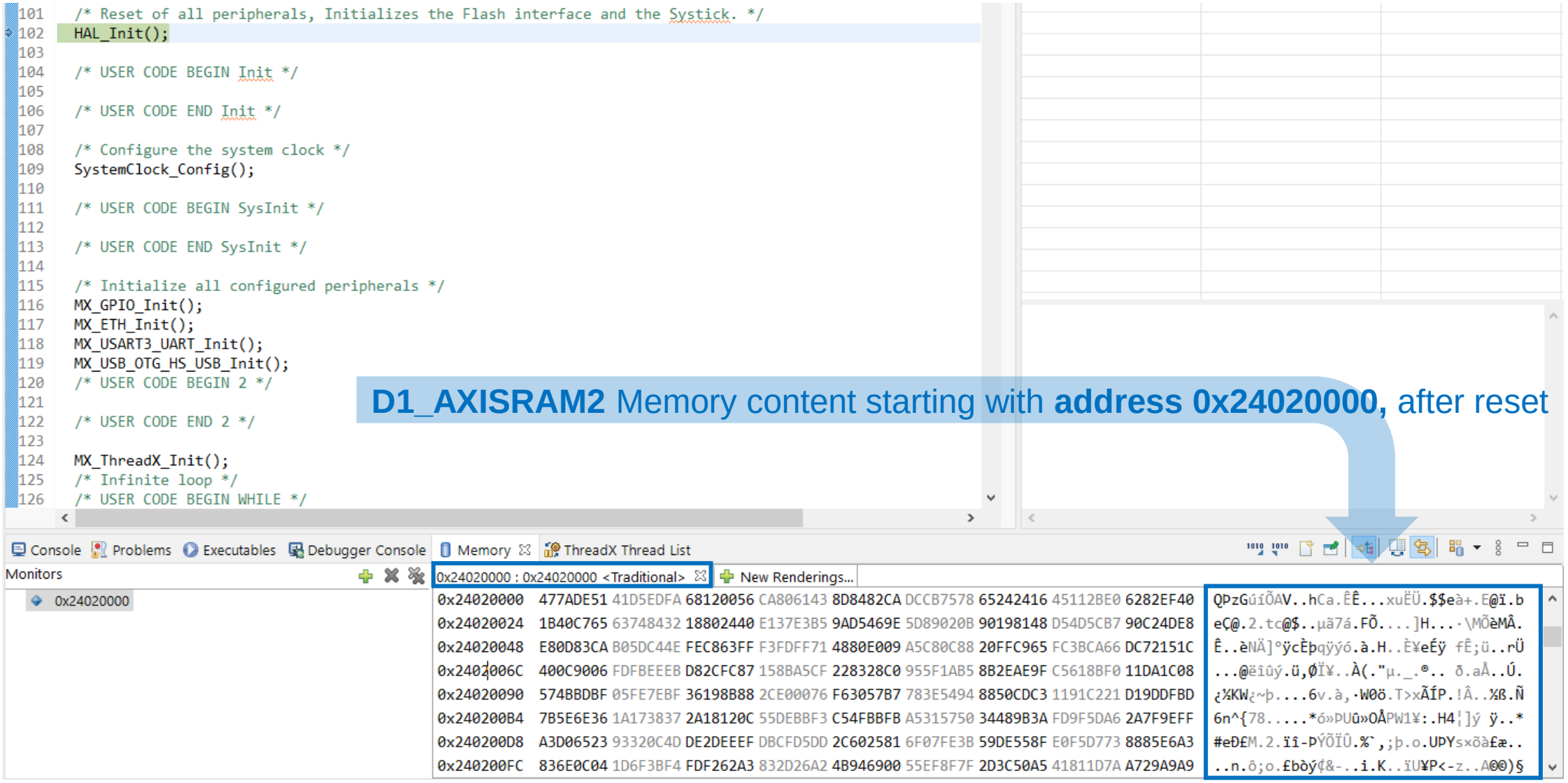


Debugging the FileX Code

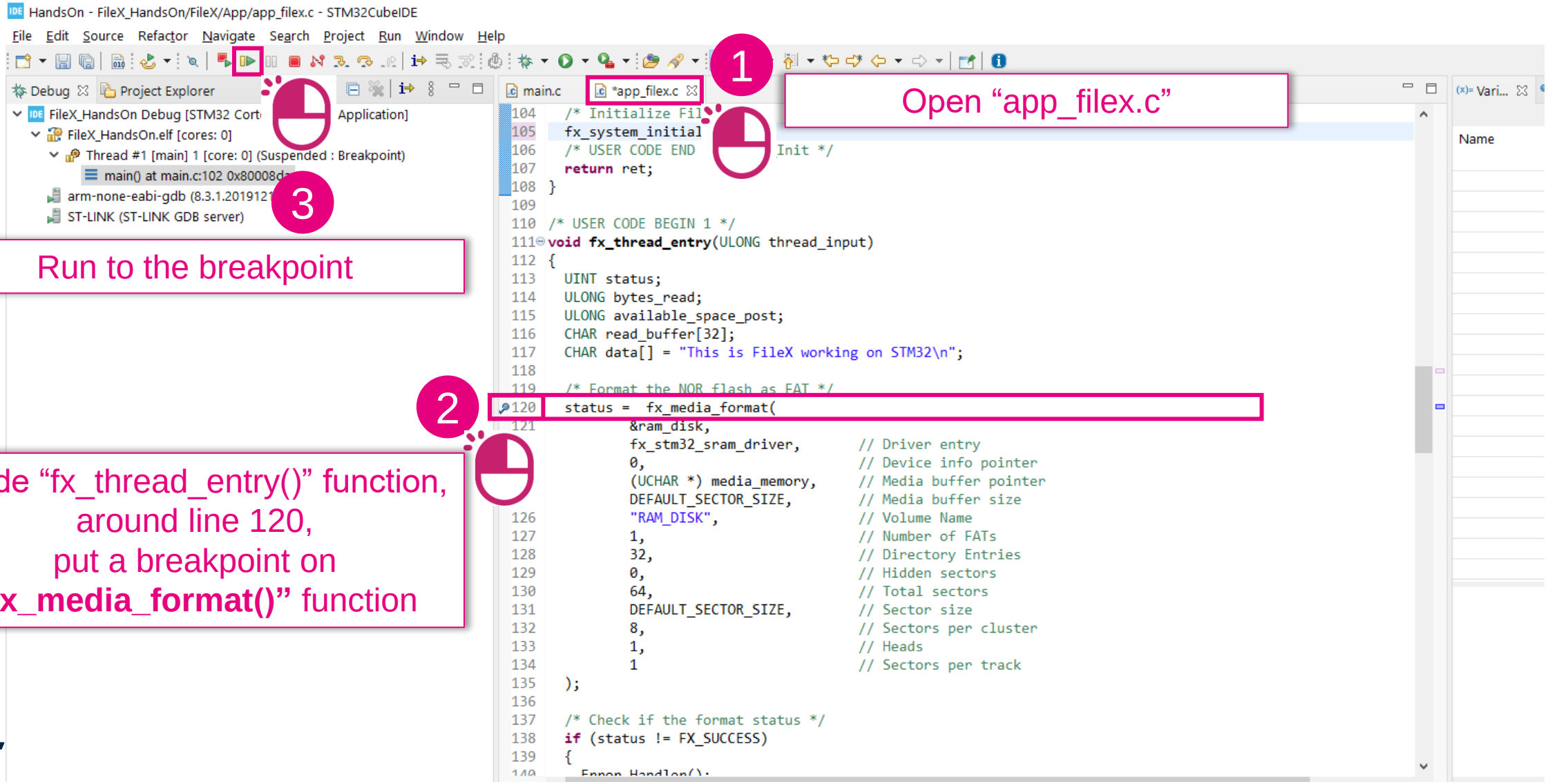
```
101   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
102   HAL_Init();
103
104   /* USER CODE BEGIN Init */
105
106   /* USER CODE END Init */
107
108   /* Configure the system clock */
109   SystemClock_Config();
110
111   /* USER CODE BEGIN SysInit */
112
113   /* USER CODE END SysInit */
114
115   /* Initialize all configured peripherals */
116   MX_GPIO_Init();
117   MX_ETH_Init();
118   MX_USART3_UART_Init();
119   MX_USB_OTG_HS_USB_Init();
120   /* USER CODE BEGIN 2 */
121
122   /* USER CODE END 2 */
123
124   MX_ThreadX_Init();
125   /* Infinite loop */
126   /* USER CODE BEGIN WHILE */
```



Debugging the FileX Code



Debugging the FileX Code



life.augmented

Debugging the FileX Code

fx_media_format () func. will format our memory where we are creating the file system.

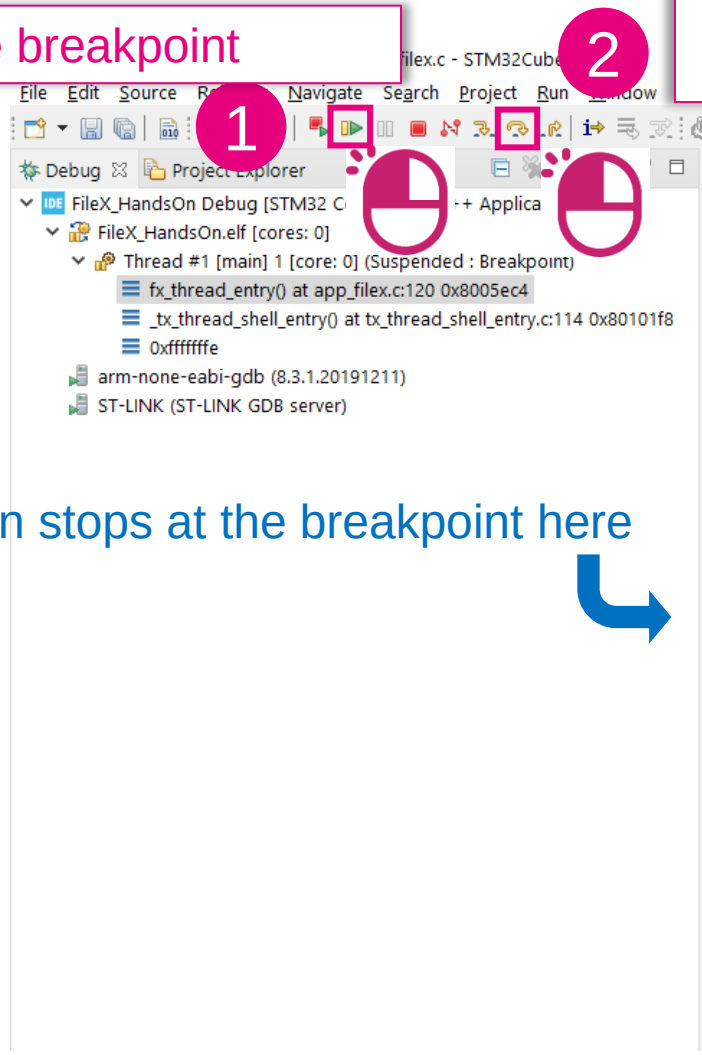
```
119   /* Format the NOR flash as FAT */
120   status =  fx_media_format(
121           &ram_disk,
122           fx_stm32_sram_driver,       // Driver entry
123           0,                          // Device info pointer
124           (UCHAR *) media_memory,     // Media buffer pointer
125           DEFAULT_SECTOR_SIZE,        // Media buffer size
126           "RAM_DISK",                 // Volume Name
127           1,                          // Number of FATs
128           32,                         // Directory Entries
129           0,                          // Hidden sectors
130           64,                         // Total sectors
131           DEFAULT_SECTOR_SIZE,        // Sector size
132           8,                          // Sectors per cluster
133           1,                          // Heads
134           1                           // Sectors per track
135   );
136
```

“RAM_DISK” is the volume name of the ram disk where we will write the file name.

Debugging the FileX Code

Run to the breakpoint

Step Over to execute the format operation



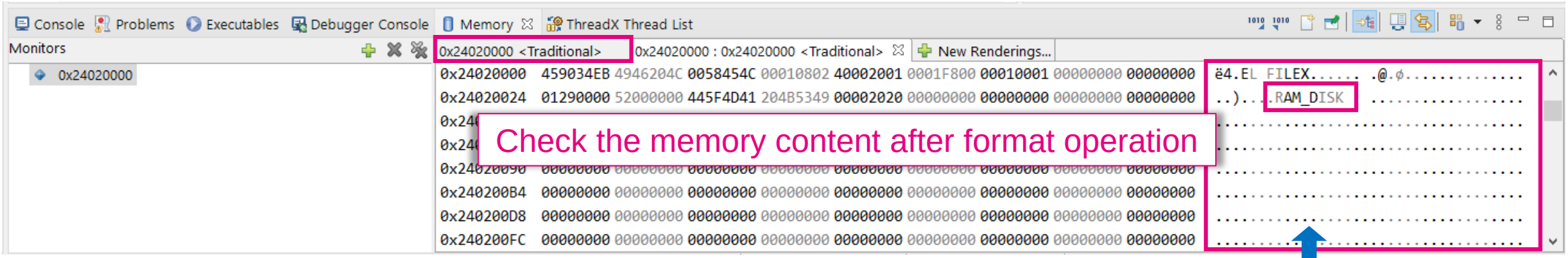
Execution stops at the breakpoint here

FileX system has already been initialized

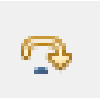
main.c *app_filex.c

```
104   /* Initialize FileX.  */
105   fx_system_initialize();
106   /* USER CODE END MX_FileX_Init */
107   return ret;
108 }
109
110 /* USER CODE BEGIN 1 */
111 void fx_thread_entry(ULONG thread_input)
112 {
113   UINT status;
114   ULONG bytes_read;
115   ULONG available_space_post;
116   CHAR read_buffer[32];
117   CHAR data[] = "This is FileX working on STM32\n";
118
119   /* Format the NOR flash as FAT */
120   status =  fx_media_format(
121           &ram_disk,
122           fx_stm32_sram_driver,       // Driver entry
123           0,                          // Device info pointer
124           (UCHAR *) media_memory,     // Media buffer pointer
125           DEFAULT_SECTOR_SIZE,        // Media buffer size
126           "RAM_DISK",                 // Volume Name
127           1,                          // Number of FATs
128           32,                         // Directory Entries
129           0,                          // Hidden sectors
130           64,                         // Total sectors
131           DEFAULT_SECTOR_SIZE,        // Sector size
132           8,                          // Sectors per cluster
133           1,                          // Heads
134           1                           // Sectors per track
135   );
136
```

Debugging the FileX Code



Volume “RAM_DISK” has been created after fx_media_format()

- So far, we have successfully formatted the media and created a FAT file system **on** our **ram disk**.
- Now we can keep stepping through the code by step over  and try to understand the main thread code:

Debugging the FileX Code

Step over until

```
143   /* Open the RAM disk.  */
144   status =  fx_media_open(&ram_disk, "RAM DISK", fx_stm32_sram_driver, 0, (VOID *) media_memory, DEFAULT_SECTOR_SIZE);
145
```

fx_media_open() opens the ram disk media for the file access before creating the file.

Step over until

```
152   /* Create a file called STM32.TXT in the root directory.  */
153   status =  fx_file_create(&ram_disk, "FXTEST.TXT");
154
```

fx_file_create() creates a file with the name “FXTEST.TXT” on the disk.

life.augmented

Debugging the FileX Code

Step over until

```
167   /* Open the test file.  */
168   status =  fx_file_open(&ram_disk, &fx_file, "FXTEST.TXT", FX_OPEN_FOR_WRITE);
169
```

fx_file_open() opens the file for **write access.**

Step over until

```
177   /* Seek to the beginning of the test file.  */
178   status =  fx_file_seek(&fx_file, 0);
179
```

fx_file_seek() seeks for the beginning of the file where we will write into

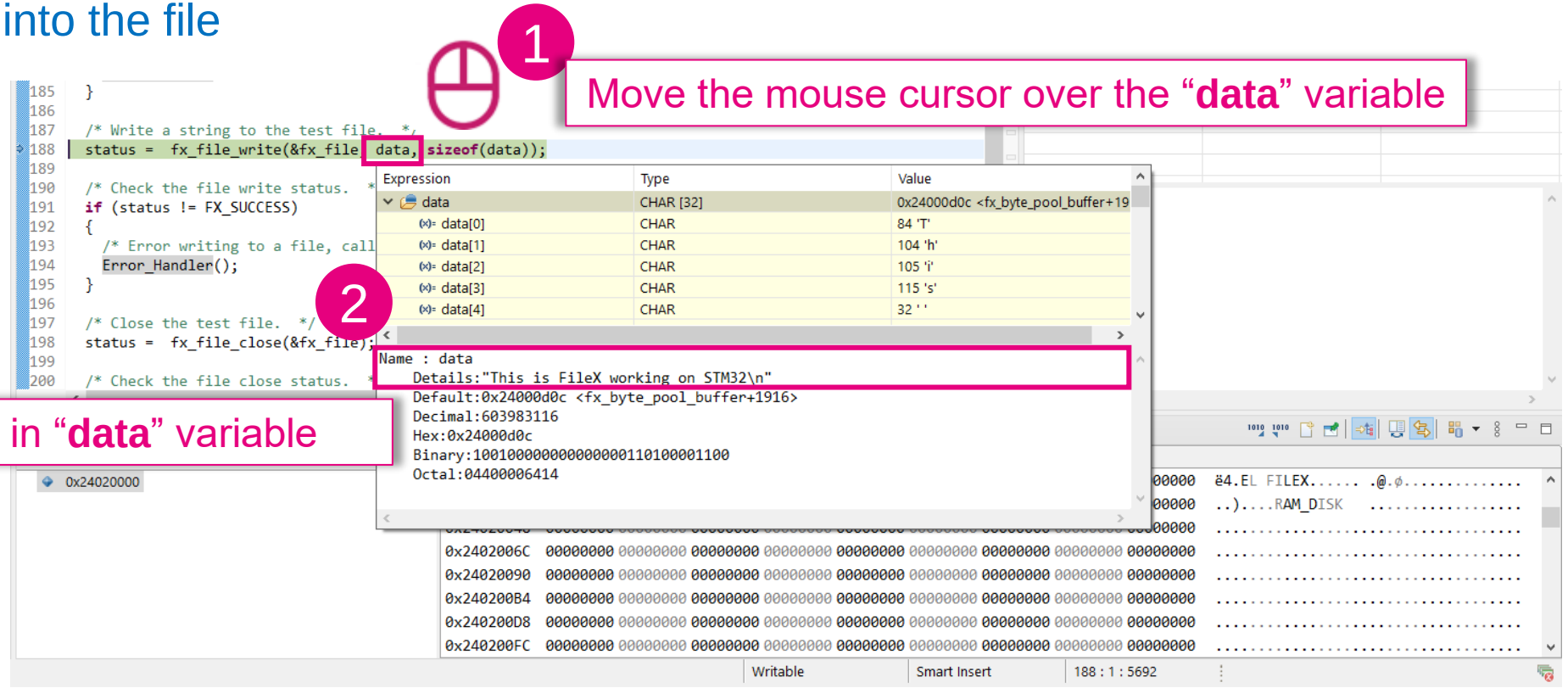
Debugging the FileX Code

Step over until

```
187   /* Write a string to the test file.  */
188   status =  fx_file_write(&fx_file, data, sizeof(data));
189
```

Writes the data string into the file

1 Move the mouse cursor over the "**data**" variable



2 Check the **string** stored in "**data**" variable

Debugging the FileX Code

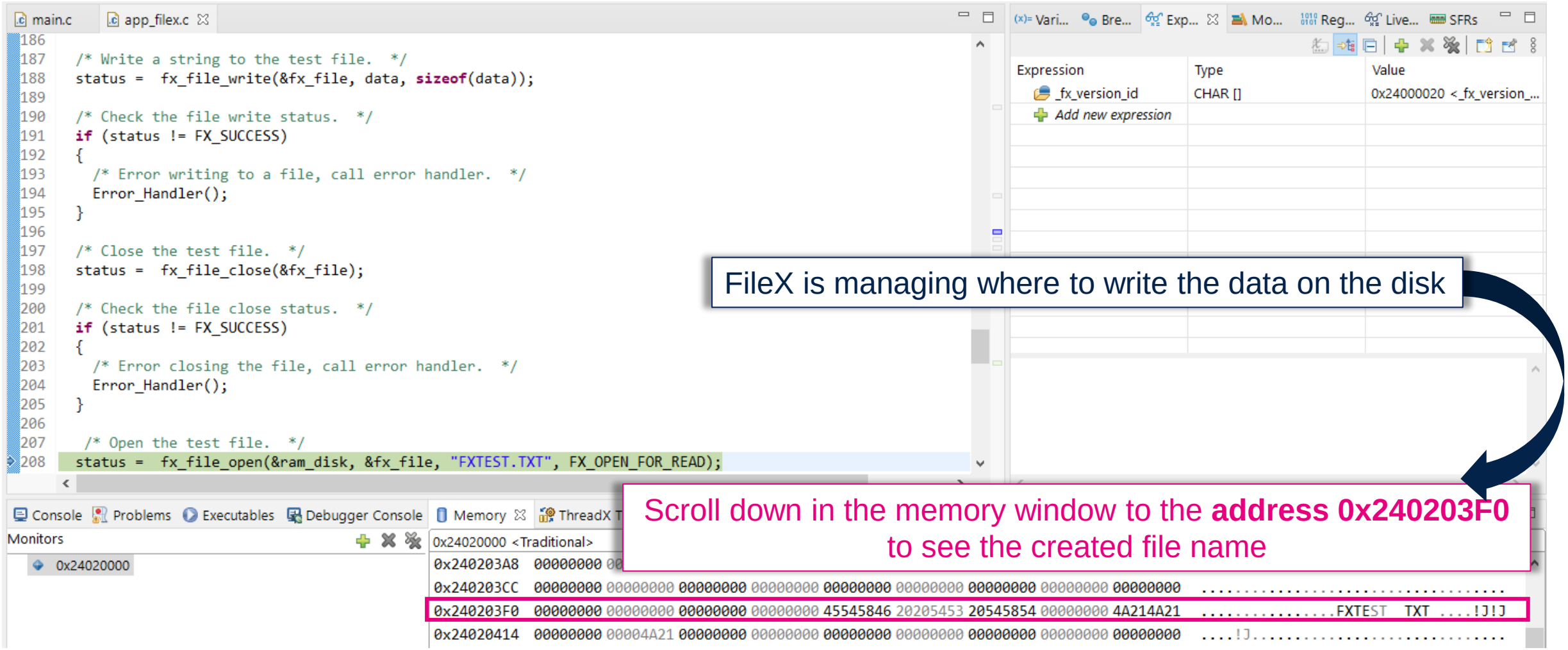
Step over until

```
197   /* Close the test file.  */
198   status =  fx_file_close(&fx_file);
199
```

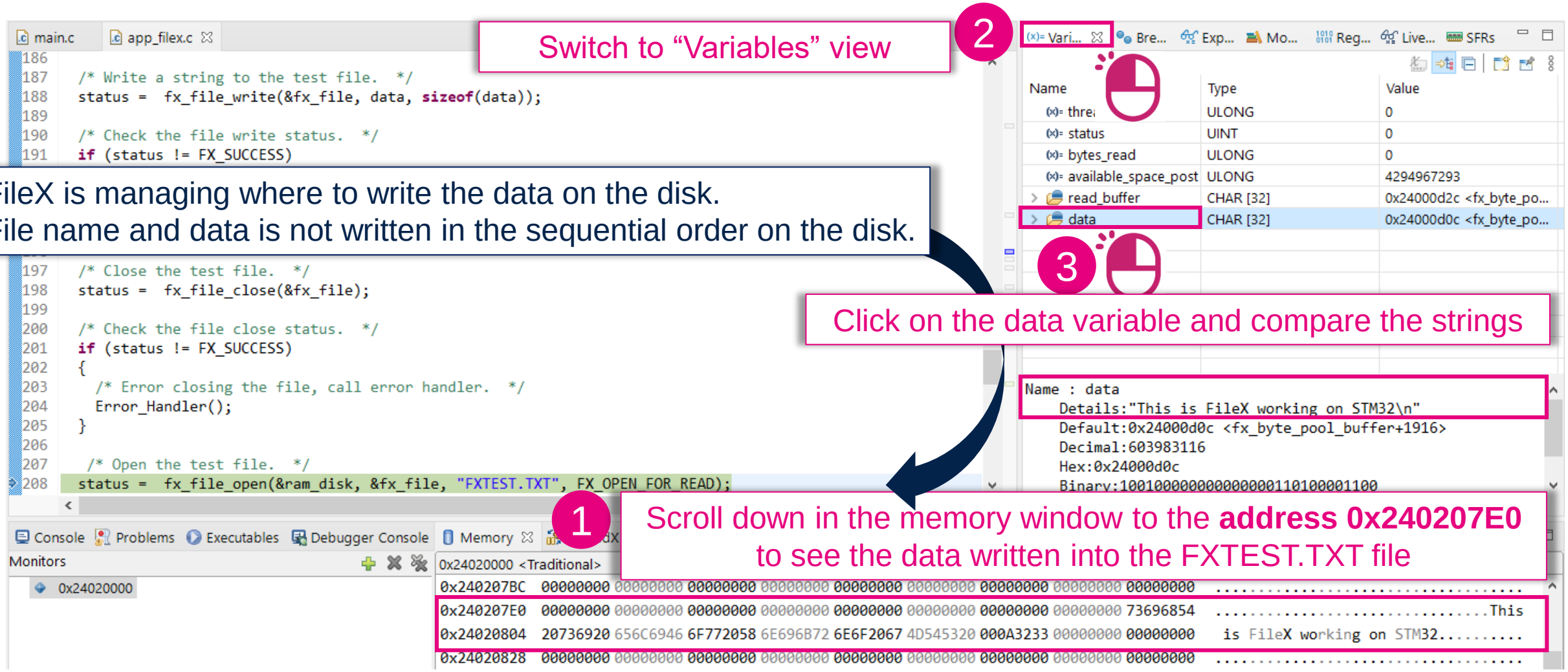
Finally, we need to close the file after completing the write operation.

- Now, let’s check the created file on the ram disk and data written into this file

Debugging the FileX Code



Debugging the FileX Code



life.augmented

Debugging the FileX Code

Step over until

```
207    /* Open the test file.  */
208    status =  fx_file_open(&ram_disk, &fx_file, "FXTEST.TXT", FX_OPEN_FOR_READ);
209
```

Opens the file for read access

Step over until

```
217    /* Seek to the beginning of the test file.  */
218    status =  fx_file_seek(&fx_file, 0);
219
```

Seeks for the beginning of the file where we will read from

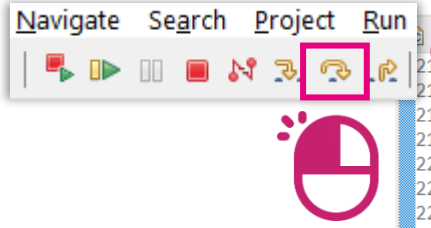
life.augmented

Debugging the FileX Code

Step over until

```
227   /* Read the first 28 bytes of the test file.  */
228   status =  fx_file_read(&fx_file, read_buffer, sizeof(data), &bytes_read);
229
```

Reads the data from the file into “read_buffer”



1

Step Over again to execute the read operation

```
217   /* Se
218   statu
219
220   /* Check the file seek status.  */
221   if (status != FX_SUCCESS)
222   {
223     /* Error performing file seek, call error handler.  */
224     Error_Handler();
225   }
226
227   /* Read the first 28 bytes of the test file.  */
228   status =  fx_file_read(&fx_file, read_buffer, sizeof(data), &bytes_read);
229
230   /* Check the file read status.  */
231   if ((status != FX_SUCCESS) || (bytes_read != sizeof(data)))
232   {
233     /* Error reading file, call error handler.  */
234     Error_Handler();
235    }
236
237   /* Close the test file.  */
238   status =  fx_file_close(&fx_file);
```

Vari... Bre... Exp... Mo... Reg... Live... SFRs

Name	Type	Value
thread_input	ULONG	0
status	UINT	0
bytes_read	ULONG	32
available_space_post	ULONG	4294967293
read_buffer	CHAR [32]	0x24000d2c <fx_byte_po...
data	CHAR [32]	0x24000d0c <fx_byte_po...

2

Click on “read_buffer” and compare the strings

```
Name : read_buffer
    Details:"This is FileX working on STM32\n"
    Default:0x24000d2c <fx_byte_pool_buffer+1948>
    Decimal:603983148
    Hex:0x24000d2c
    Binary:10010000000000000110100101100
```

Console Problems Executables Debugger Console | Memory ThreadX Thread List

Monitors: 0x24020000

0x24020000 <Traditional> | 0x24020000 : 0x24020000 <Traditional> | New Renderings...

```
0x240207BC  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ................................
0x240207E0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 73696854  ............................This
0x24020804  20736920 656C6946 6F772058 6E696B72 6E6F2067 4D545320 000A3233 00000000  is FileX working on STM32.......
0x24020828  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ................................
```

life.augmented

Debugging the FileX Code

Step over until

```
237   /* Close the test file.  */
238   status =  fx_file_close(&fx_file);
239
```

Finally, we need to close the file after completing the read operation.

```
247   /* Get the available usable space, after the file has been created */
248   status =  fx_media_space_available(&ram_disk, &available_space_post);
249
```

If you need, you can also check the available space in the disk after the file create and write operations.

life.augmented

Conclusion

- We have seen how to implement FAT file system using X-CUBE-AZURE on the internal SRAM.

- Using STM32CubeMX, it is also possible to implement the same on different memories like uSD card or NOR flash, etc. by adding related drivers for these memories into the project.

Thank you

life.augmented