life.augmented

STM32 AZURE RTOS workshop

ThreadX

Introduction

- The purpose of this session is to give you a first overview of how we support AZURE-RTOS ThreadX in the STM32Cube ecosystem

Agenda

- Prepare project for CubeMX and CubeIDE
- Add ThreadX in CubeMX
- Add ThreadX into CubeIDE project
- Use TraceX
- Use multiple Threads with preemption priorities
- CMSIS_OS2 optional layer*
- ThreadX features overview

Links

Useful links

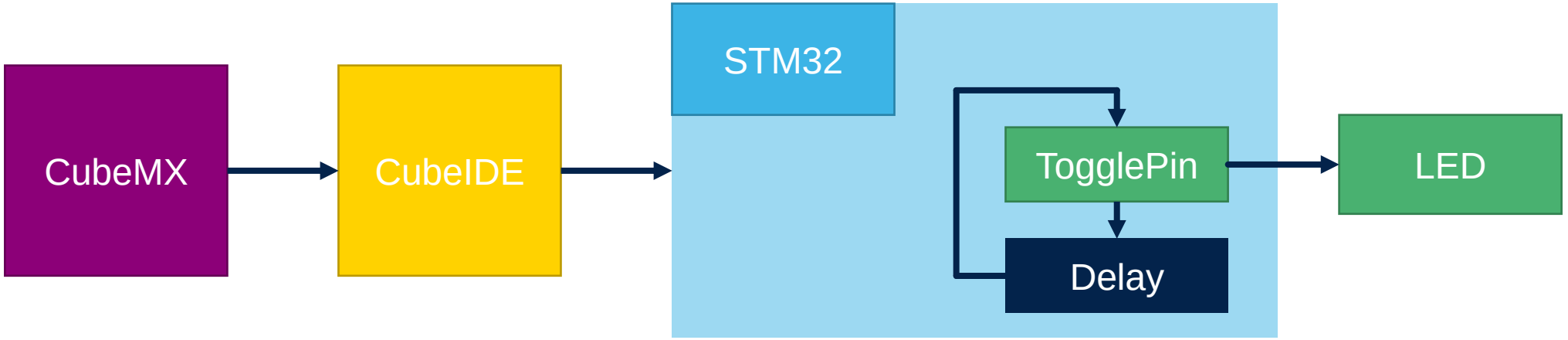
- ThreadX user guide https://docs.microsoft.com/en-us/azure/rtos/threadx/about-this-guide
- ThreadX api description https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter4
- ThreadX cheatsheet https://rristm.github.io/stm32_threadx

- Github: https://github.com/STMicroelectronics/x-cube-azrtos-h7

Step 1 : Start led blink from scratch

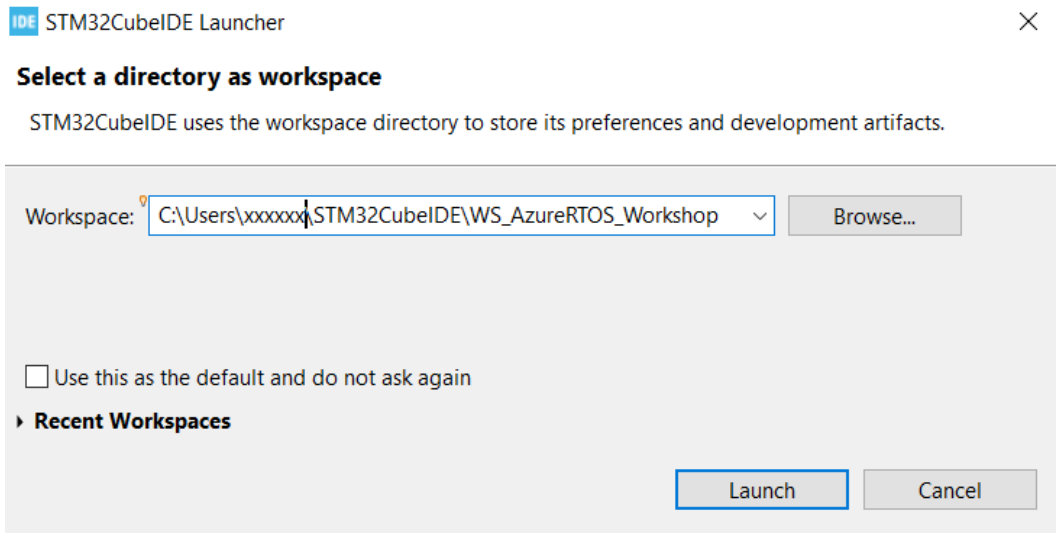
Purpose

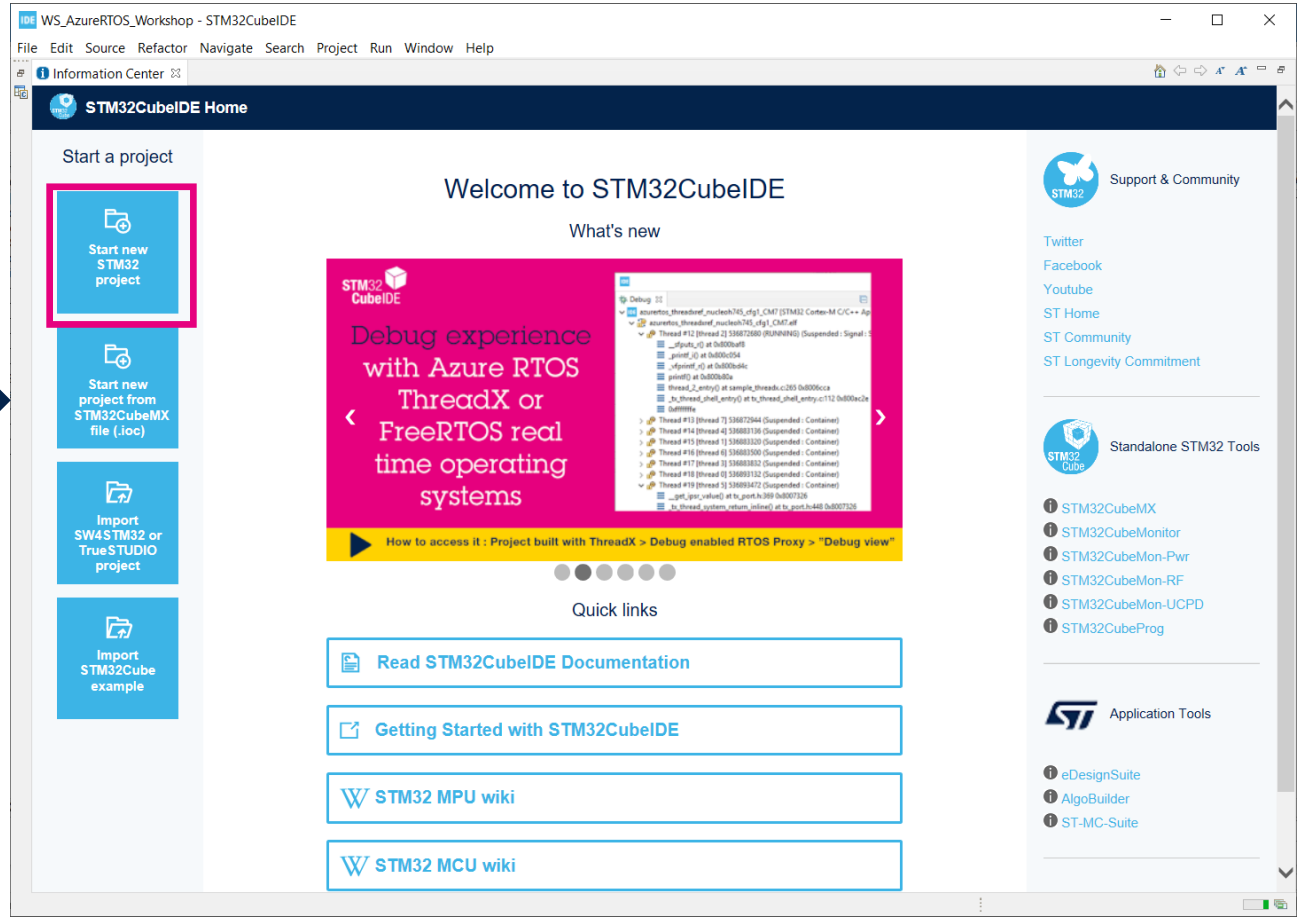
- This short hands-on will show you how to start a project from scratch
- The point is to a get first running led blink in few clicks
- We will then use this base to go further with ThreadX (no RTOS yet)



Launch STM32CubeIDE

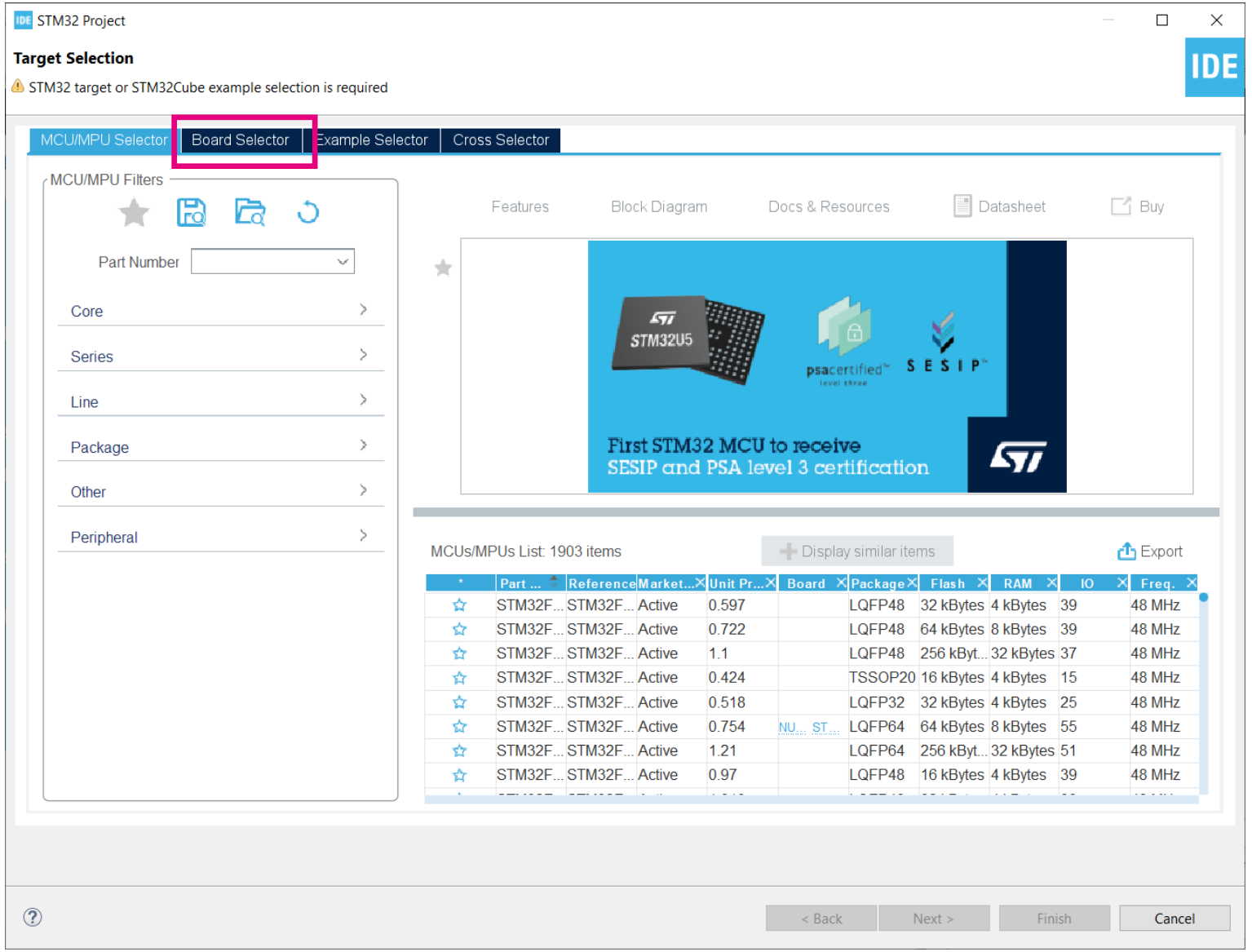
- Launch CubeIDE
- Select a workspace, click on Launch





Start a new project

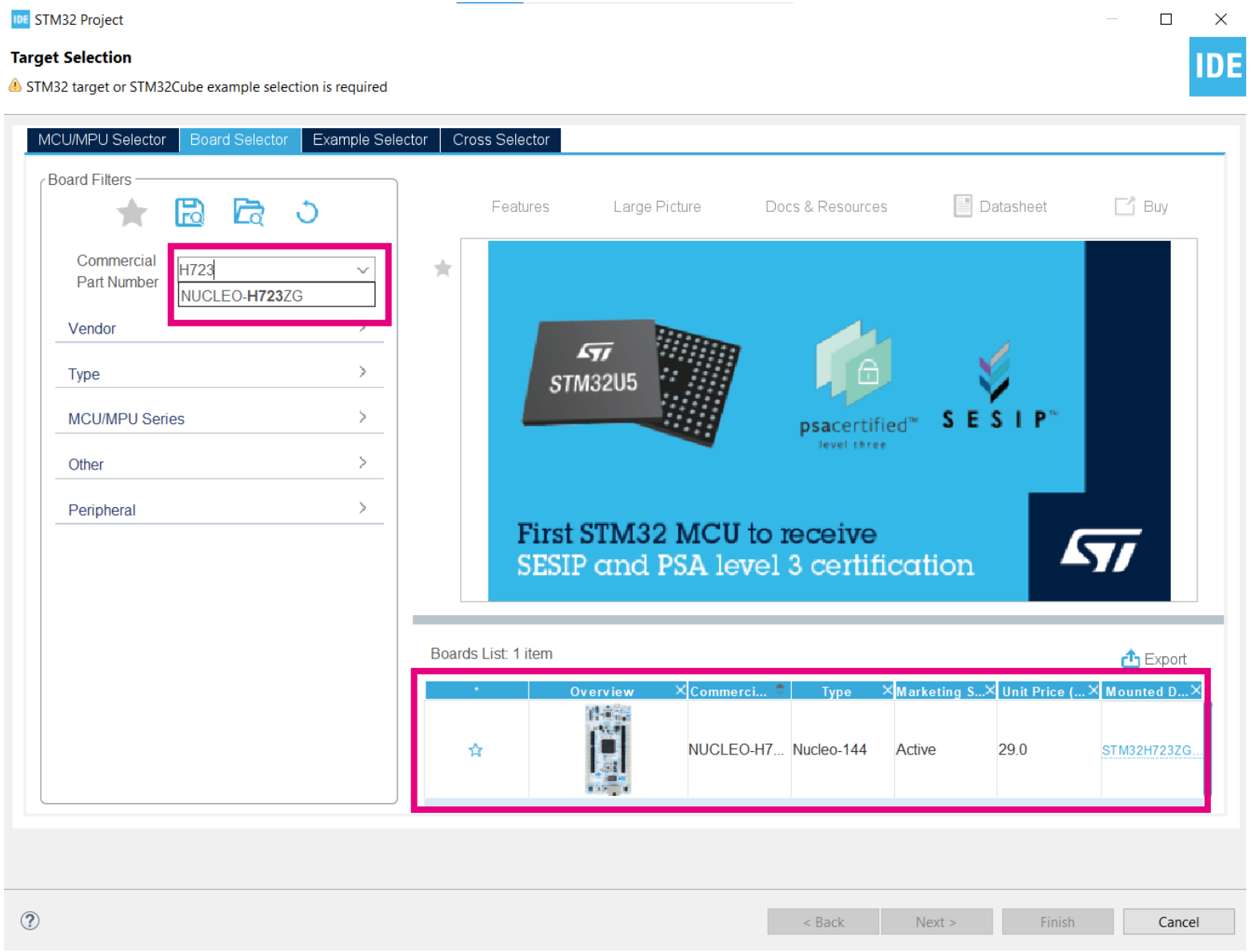
- Click on board selector



life.augmented

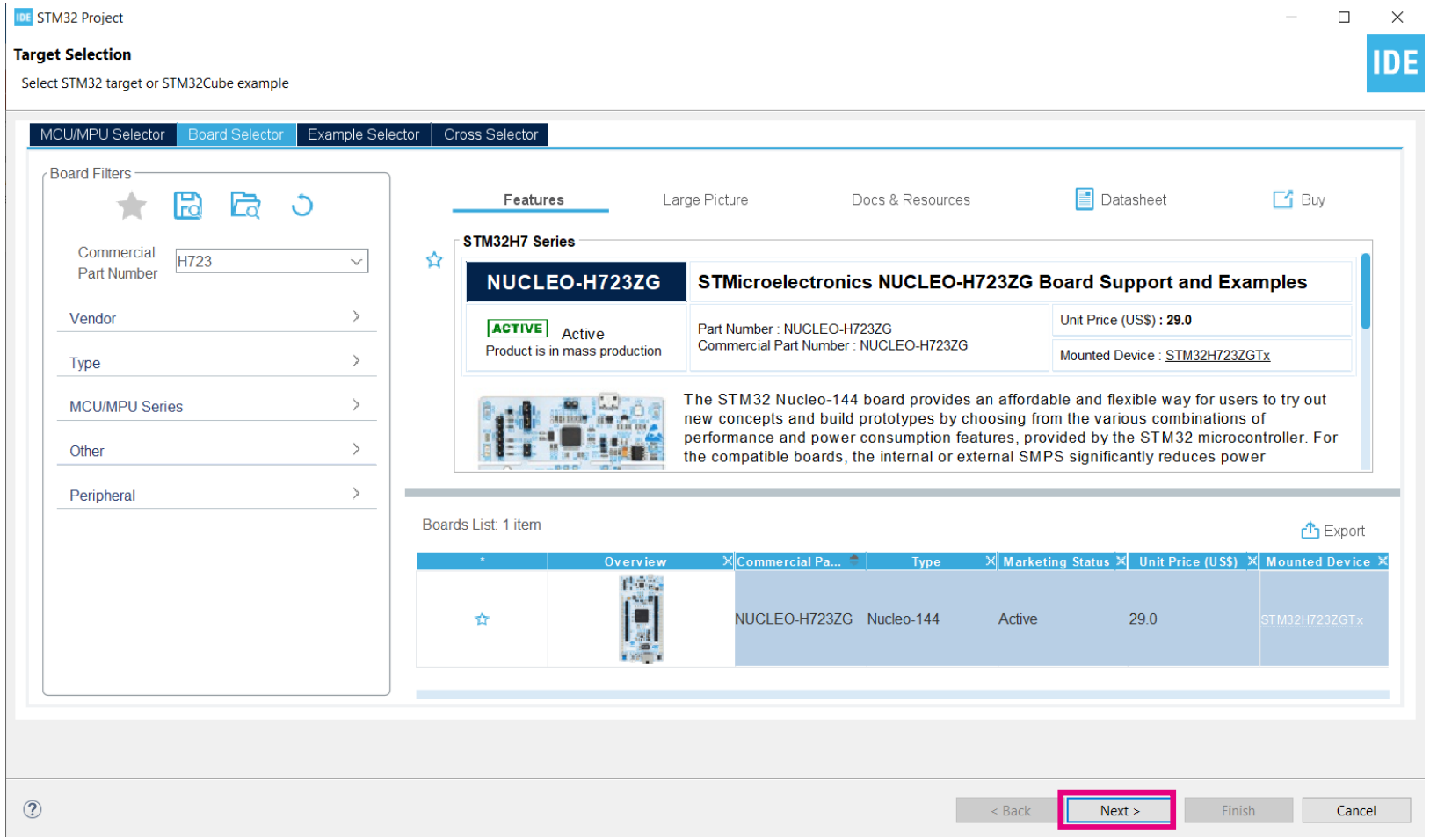
Look for Nucleo H723

- Type H723 in the search box
- Then click on Nucleo-H723ZG



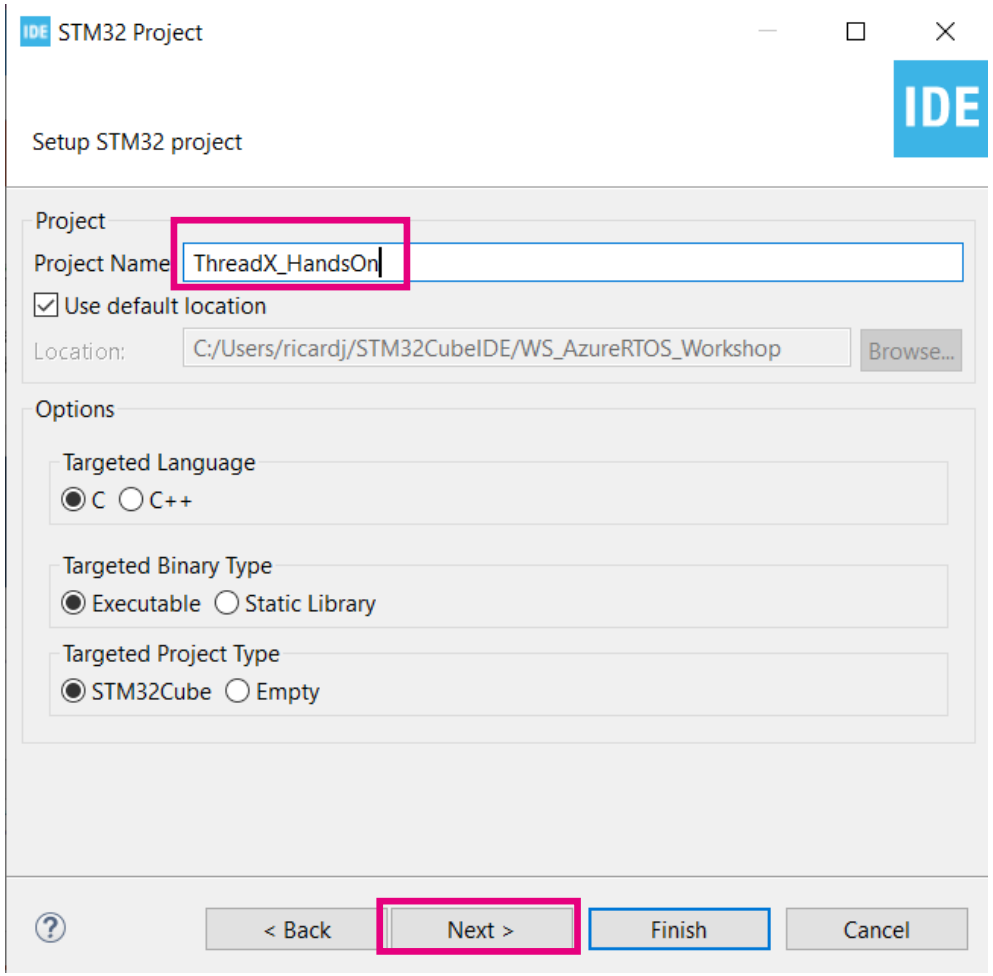
Check proposed board

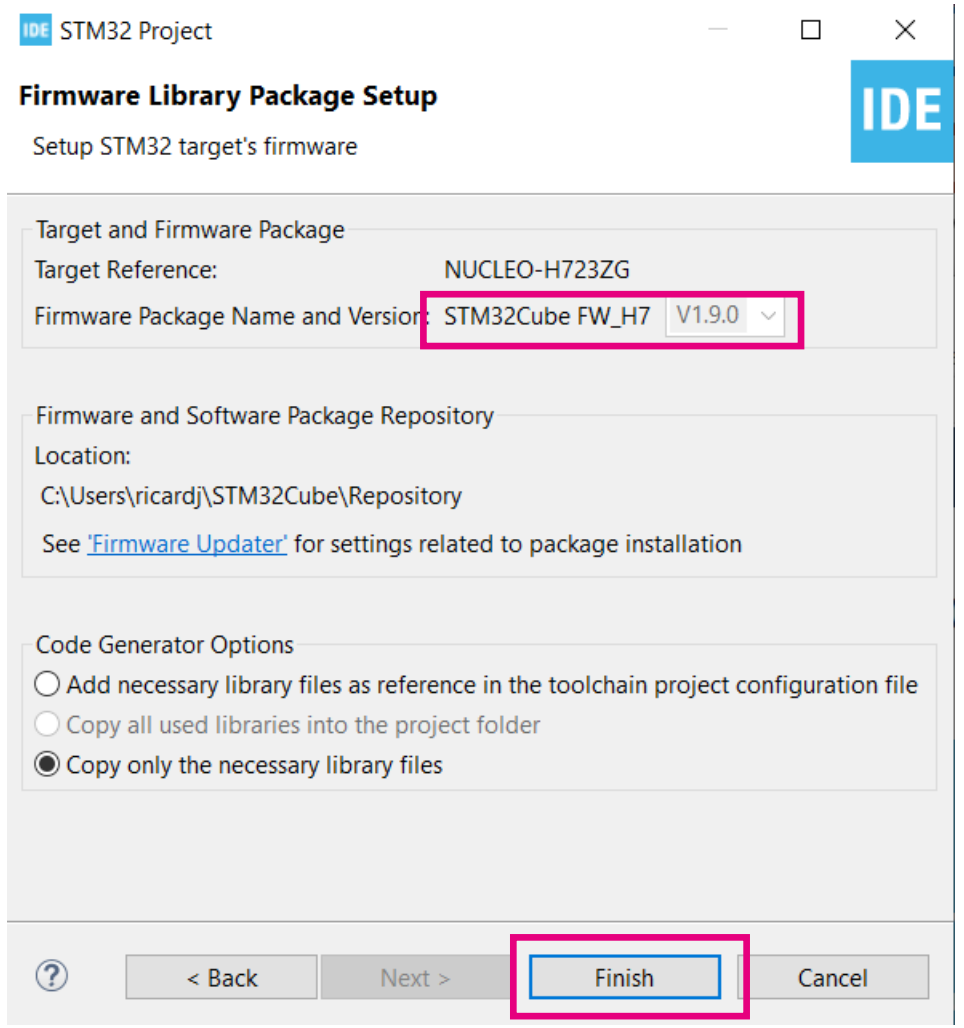
- Click next



Setup the project name

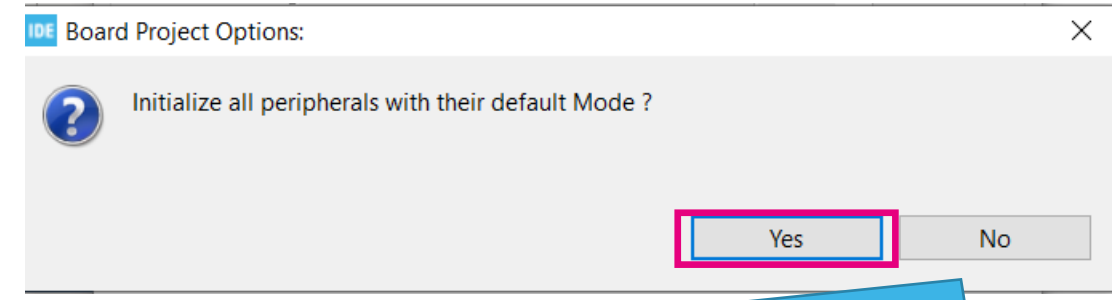
- Type the project name then next and check FW version



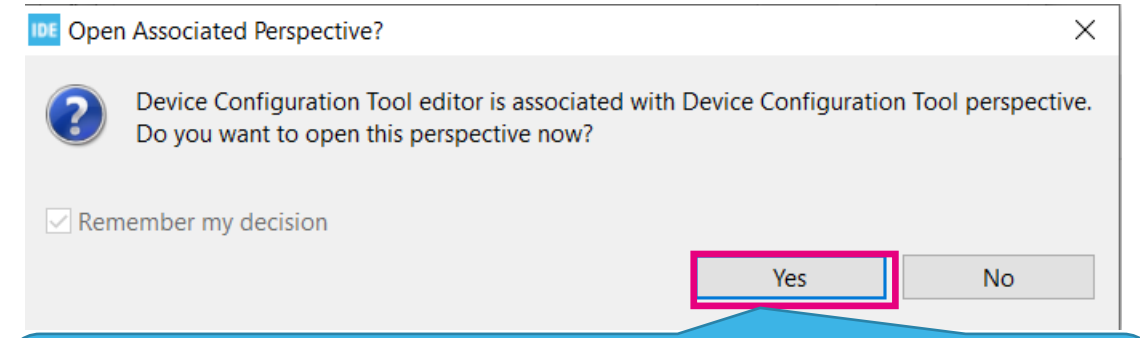


Finish the project setup

- In pop-up asking of peripherals initialization
- Click **Yes**
- Then second popup to ask to open device configuration perspective.
- Click **Yes**



This will setup all GPIOs to correct mode



This will create your project structure with all needed files and open the STM32CubeMX embedded in STM32CubeIDE

What you get after few clicks

In project explorer you get all the project structure

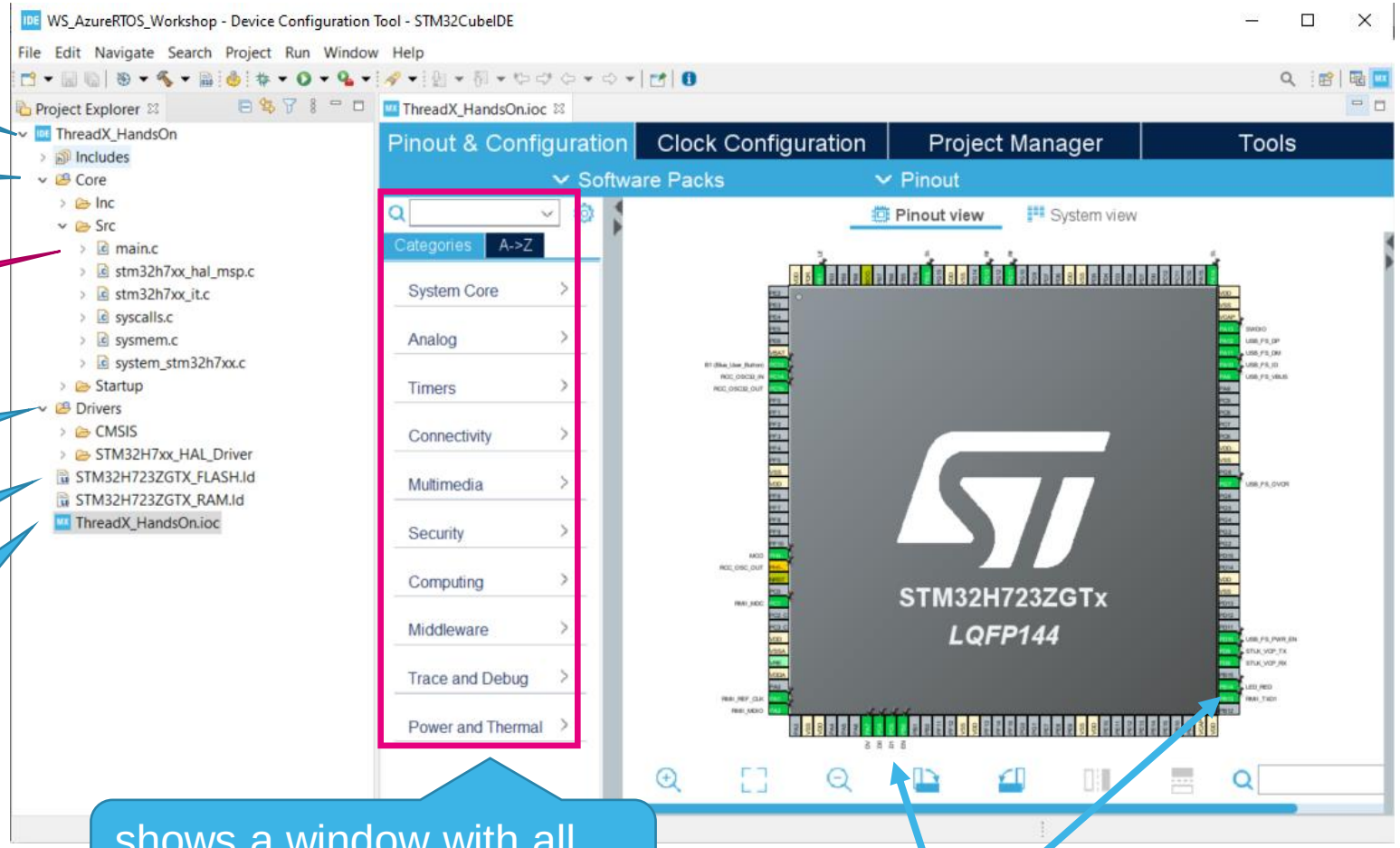
Core contains the specific project files

Our main.c

Drivers contains HAL and CMSIS

STM32H723GTX_FLASH.ld is the linker file. Handling placement in STM32 memory

Ioc file is the one opened here in the embedded STM32CubeMX



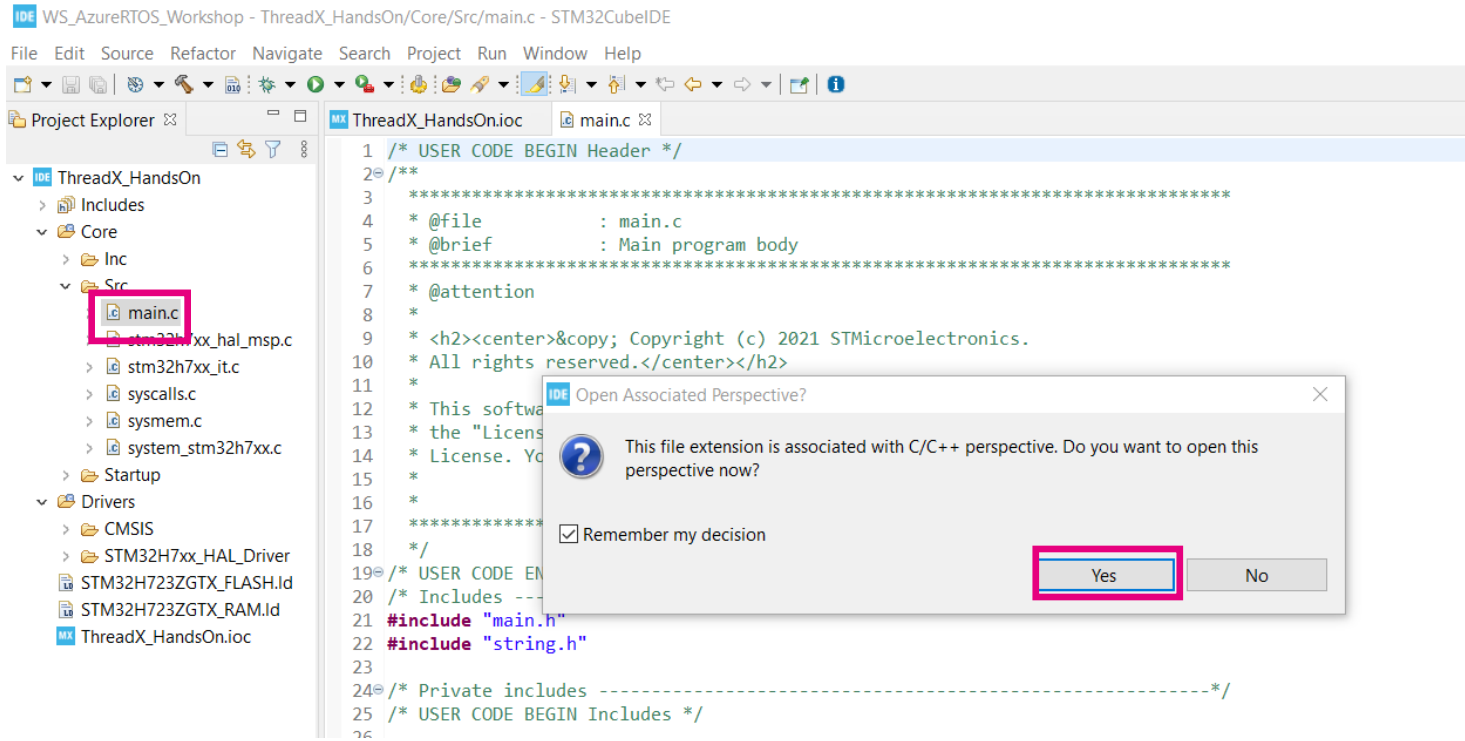
shows a window with all configurable peripherals

All GPIOs already setup according to this specific board

Edit main.c to add your code

- Look for main.c and double click
- Answer yes to the open perspective
- Then go down to the main() function and find the while(1) loop line 129

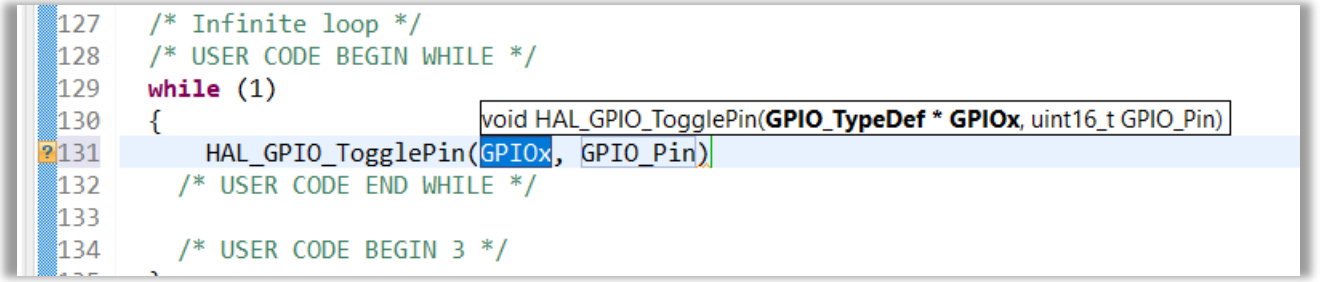
```
/* USER CODE BEGIN WHILE */
 while (1)
 {
   /* USER CODE END WHILE */

   /* USER CODE BEGIN 3 */
 }
```



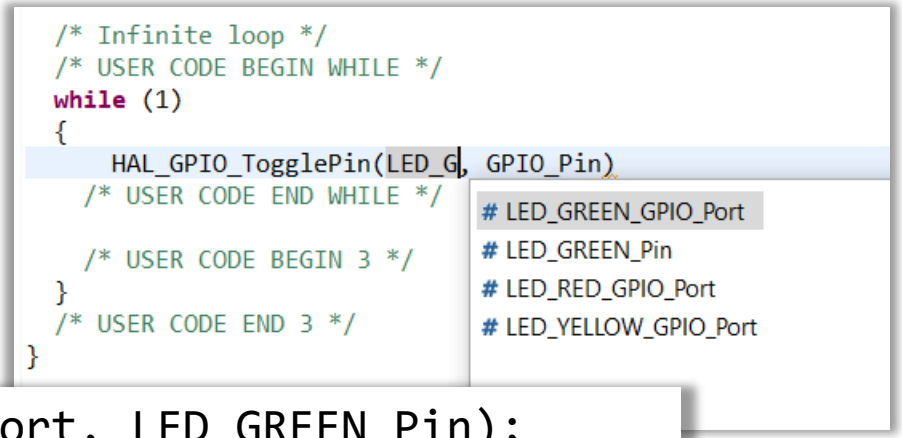
Adding LED toggling using code completion

Core\Src \main.c

- Just before comment `/* USER CODE END WHILE */` type:
- HAL_GPIO_T and the CTRL-Space bar you should get:



- First argument LED_GREEN_GPIO_Port (type LED CTRL+Space)
- Second argument LED_GREEN_Pin
- Add the final ';'




```
HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
```

life.augmented

HAL GPIO Toggle

- Each time HAL_GPIO_TogglePin is called
  - LED will turn on/off

- Without delay impossible to see in run
  - Only in debug stepping

Core\Src \main.c

Adding the delay between each LED toggle

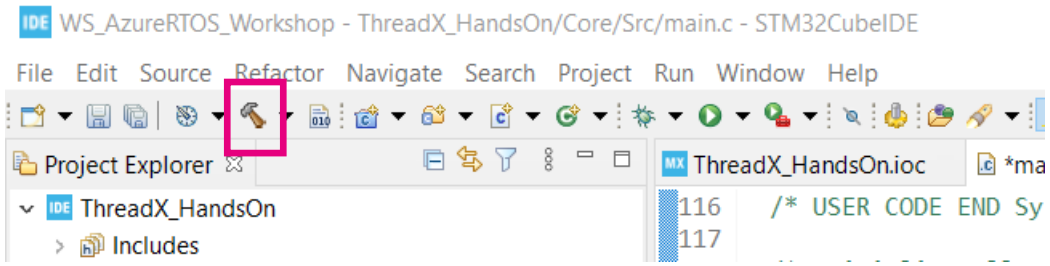
- Starting from

```
126
127   /* Infinite loop */
128   /* USER CODE BEGIN WHILE */
129   while (1)
130   {
131       HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
132
133     /* USER CODE END WHILE */
134
```

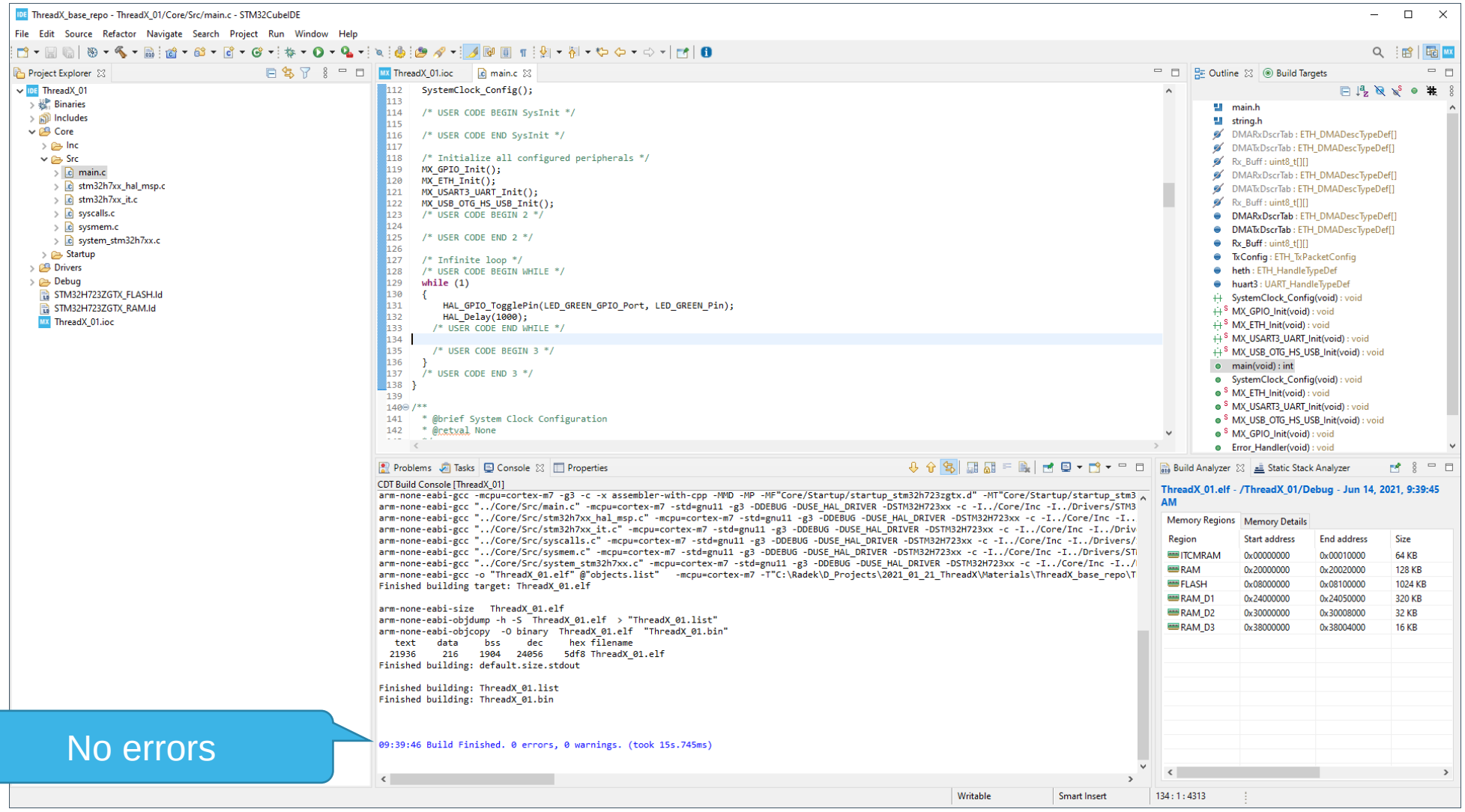
- Type HAL_Delay
- Choose a delay in ms : 1000
- Add final semicolon
- Now compile the project using the hammer

```
128   /* USER CODE BEGIN WHILE */
129   while (1)
130   {
131       HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
132       HAL_Delay(1000);
133     /* USER CODE END WHILE */
134
135     /* USER CODE BEGIN 3 */
136   }
```

```
HAL_Delay(1000);
```



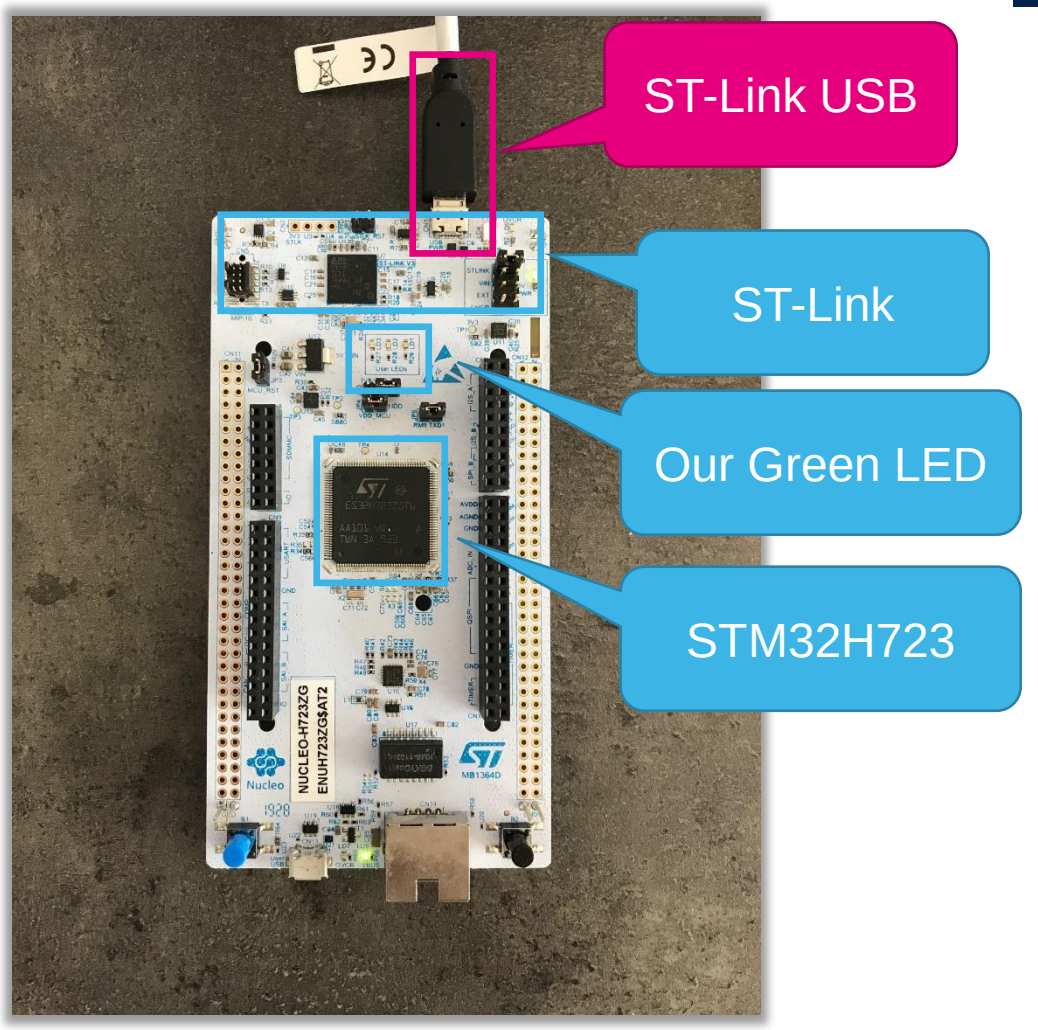
Compilation result



No errors

Connect the board to the computer

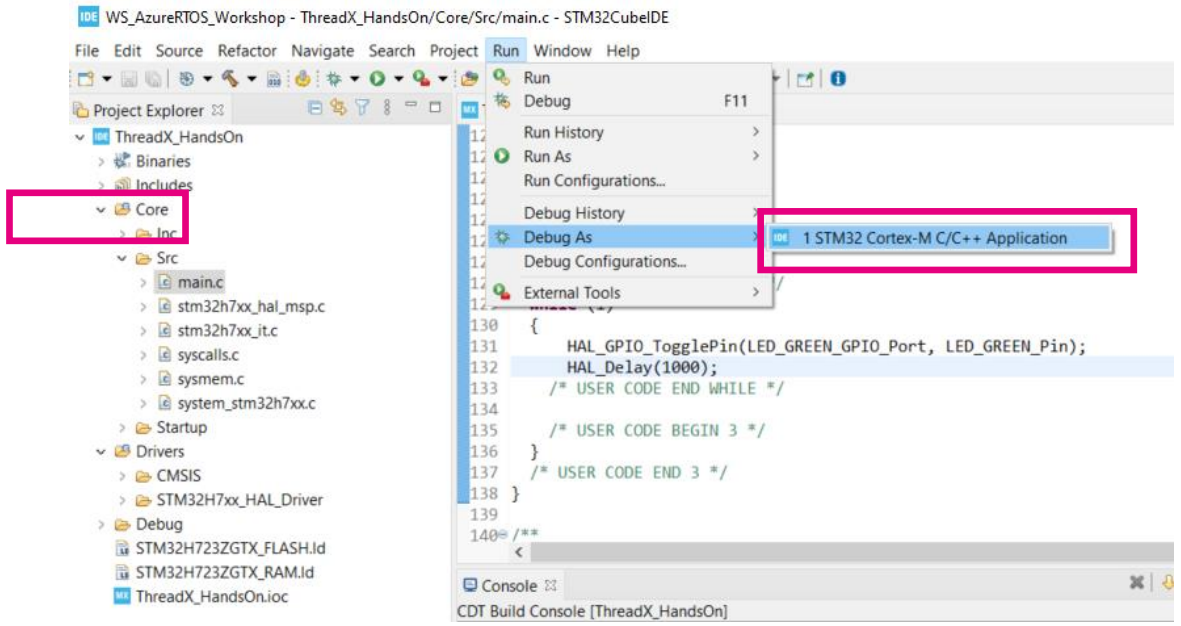
- Time to transfer firmware to the board
- First connect the Nucleo board to the computer

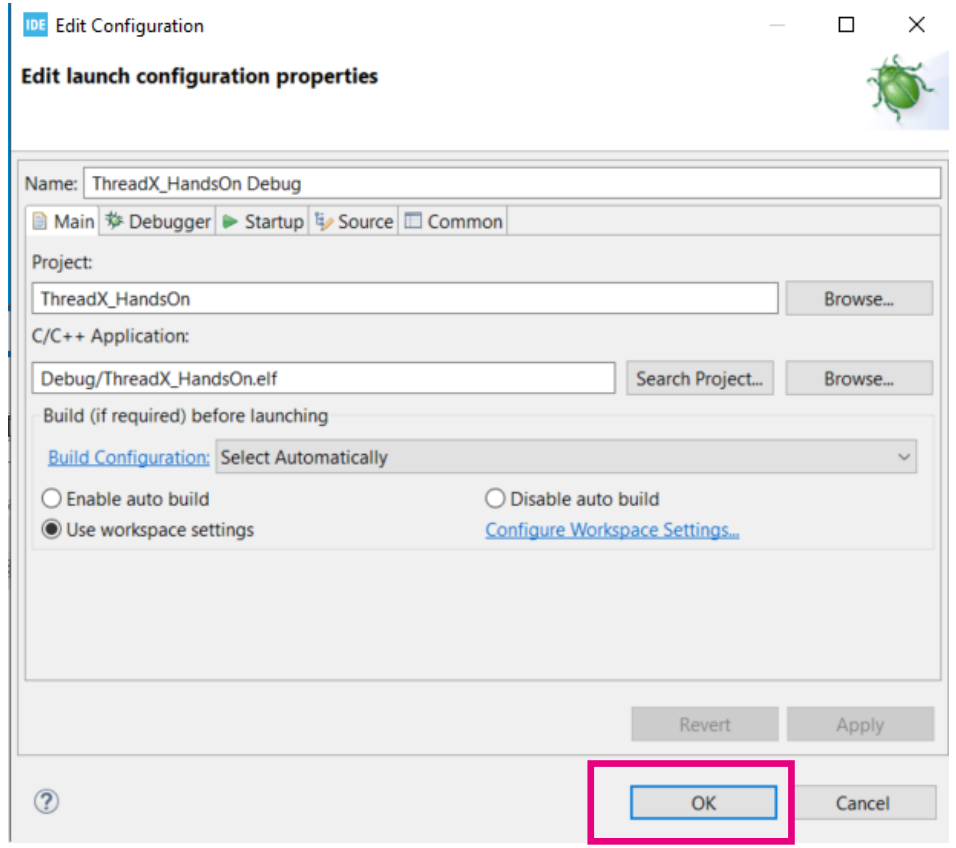


life.augmented

Download firmware and Debug on target

- First click on the project name
- Then menu Run/Debug As/STM32 …

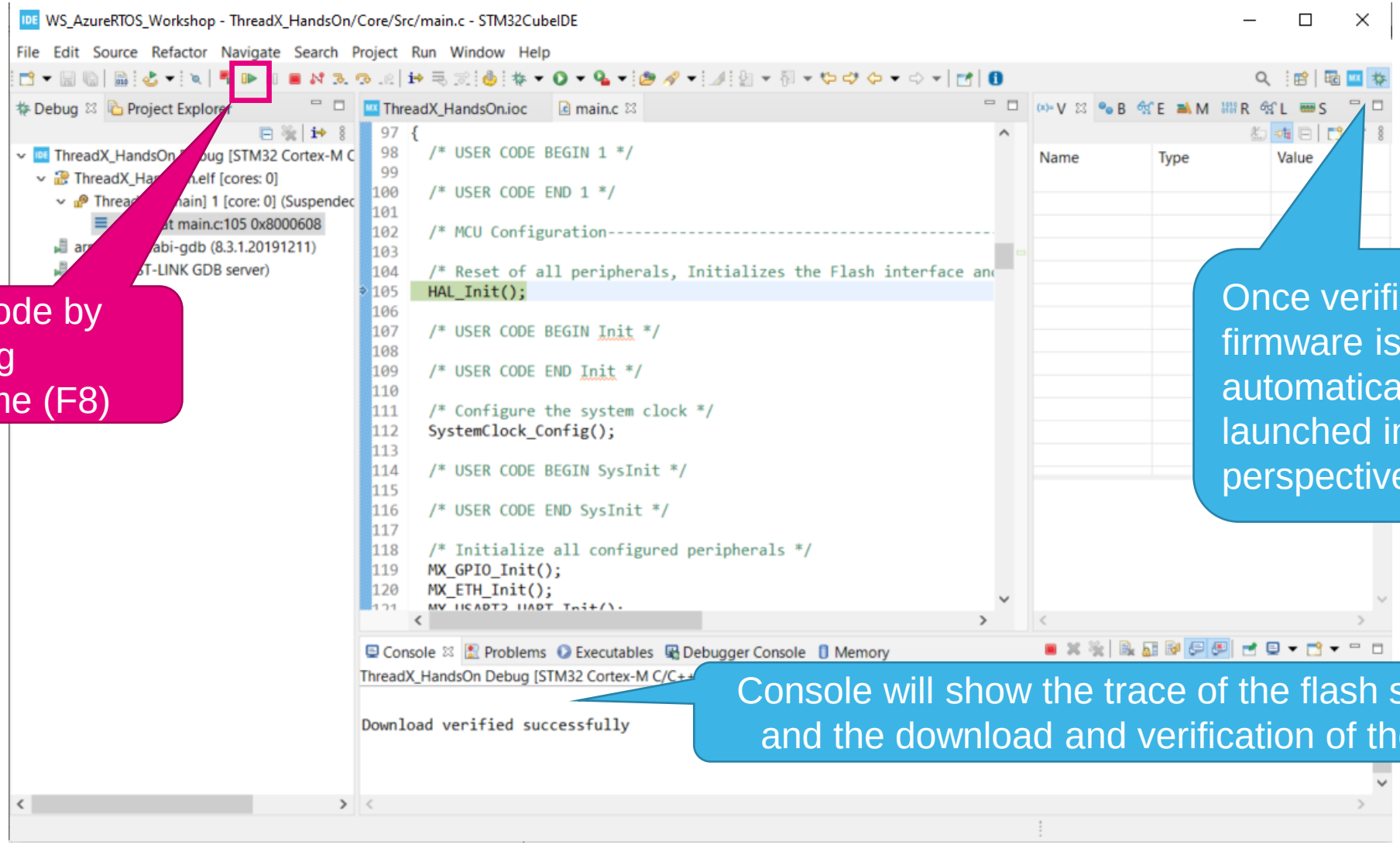




- In Edit Configuration window click OK

life.augmented

LED is Blinking

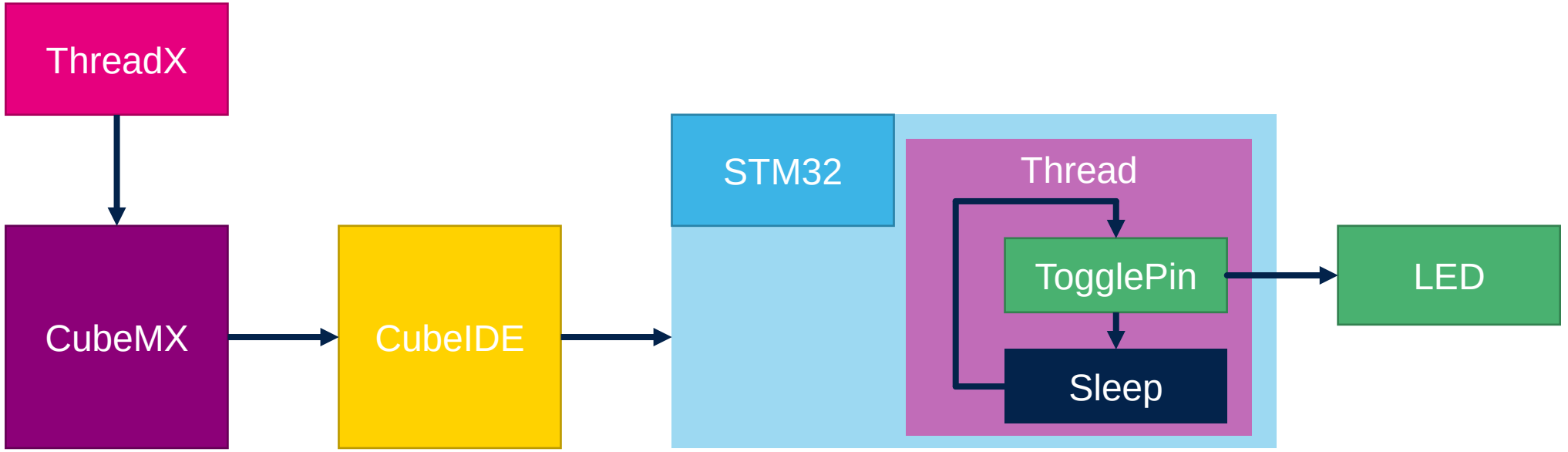


- To exit debugger click on ■ or CTRL-F2

Step 2 : Create first thread

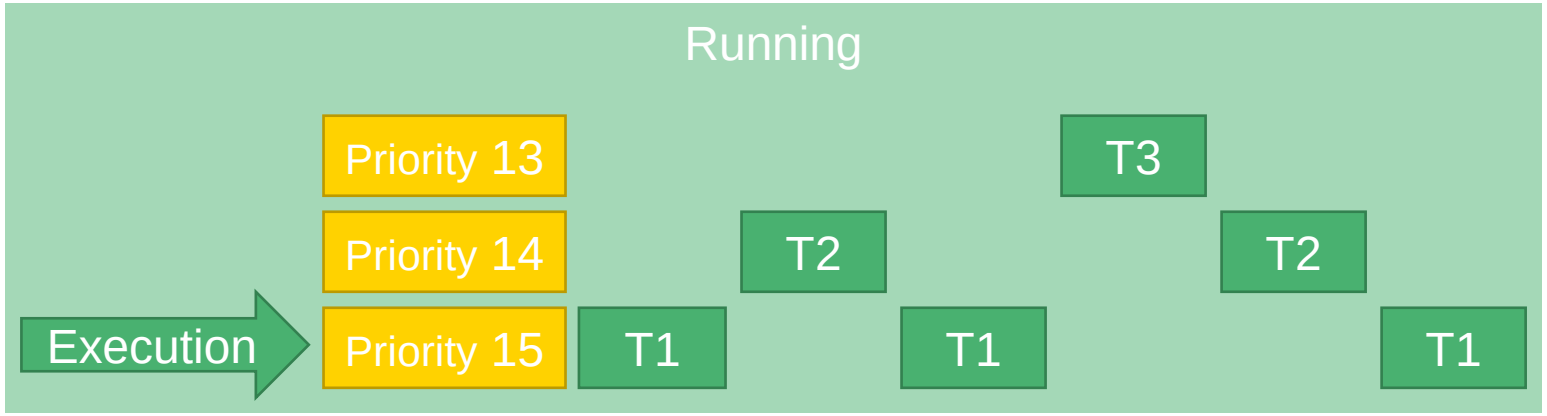
Step 2 : Create first thread

- First step is to enable ThreadX in STM32CubeMX
- We want now to blink the LED inside a ThreadX thread
- This will update the project structure to support ThreadX
- Then we will be able to simply create a thread



RTOS Thread

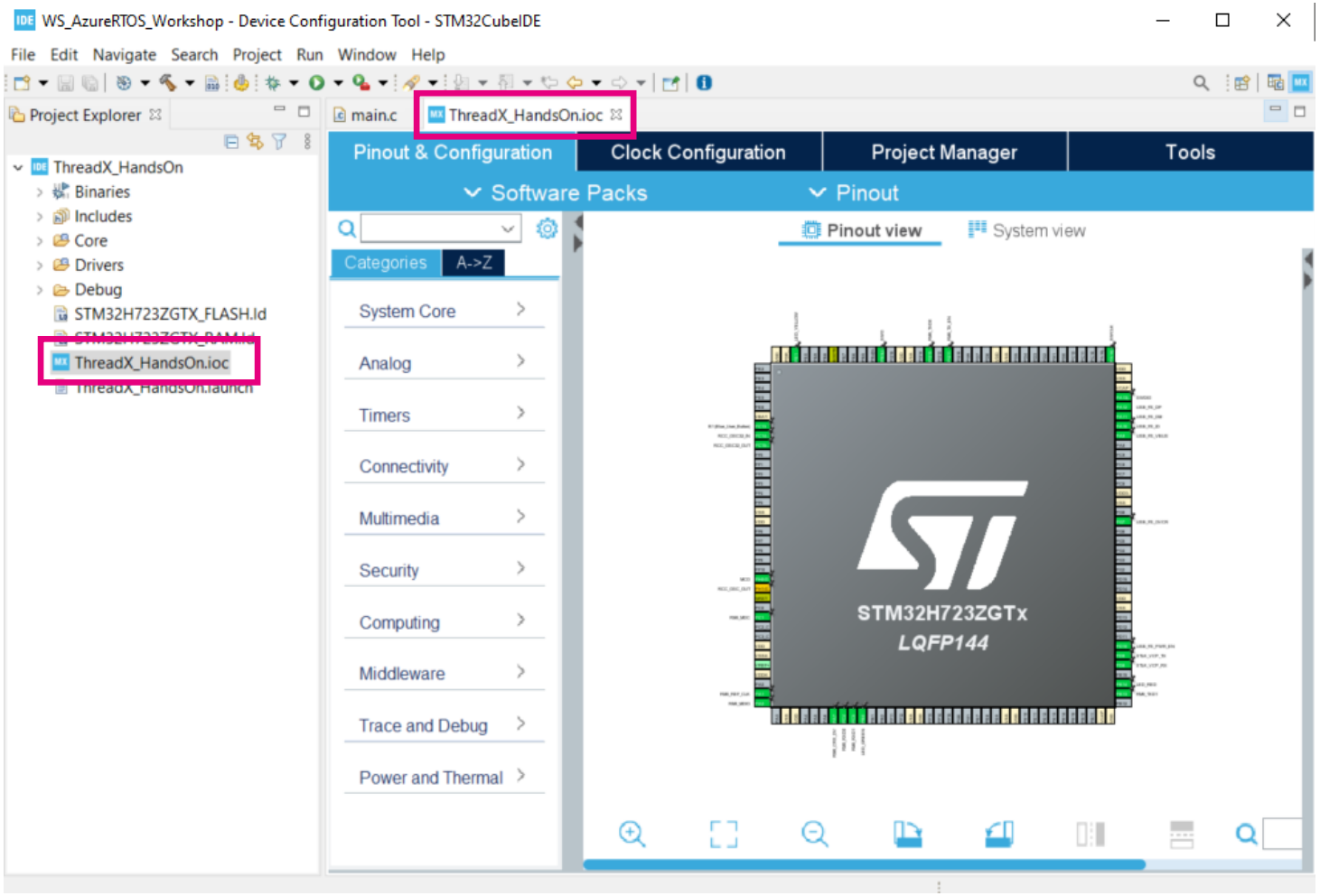
- RTOS runs in threads
  - At least one thread for user program needed (If no RTOS runs in idle)
- Preemption priority (higher priority thread run, like IRQ)



- https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter3#thread-execution-1
- https://rristm.github.io/stm32_threadx/show/thredx_thread.md

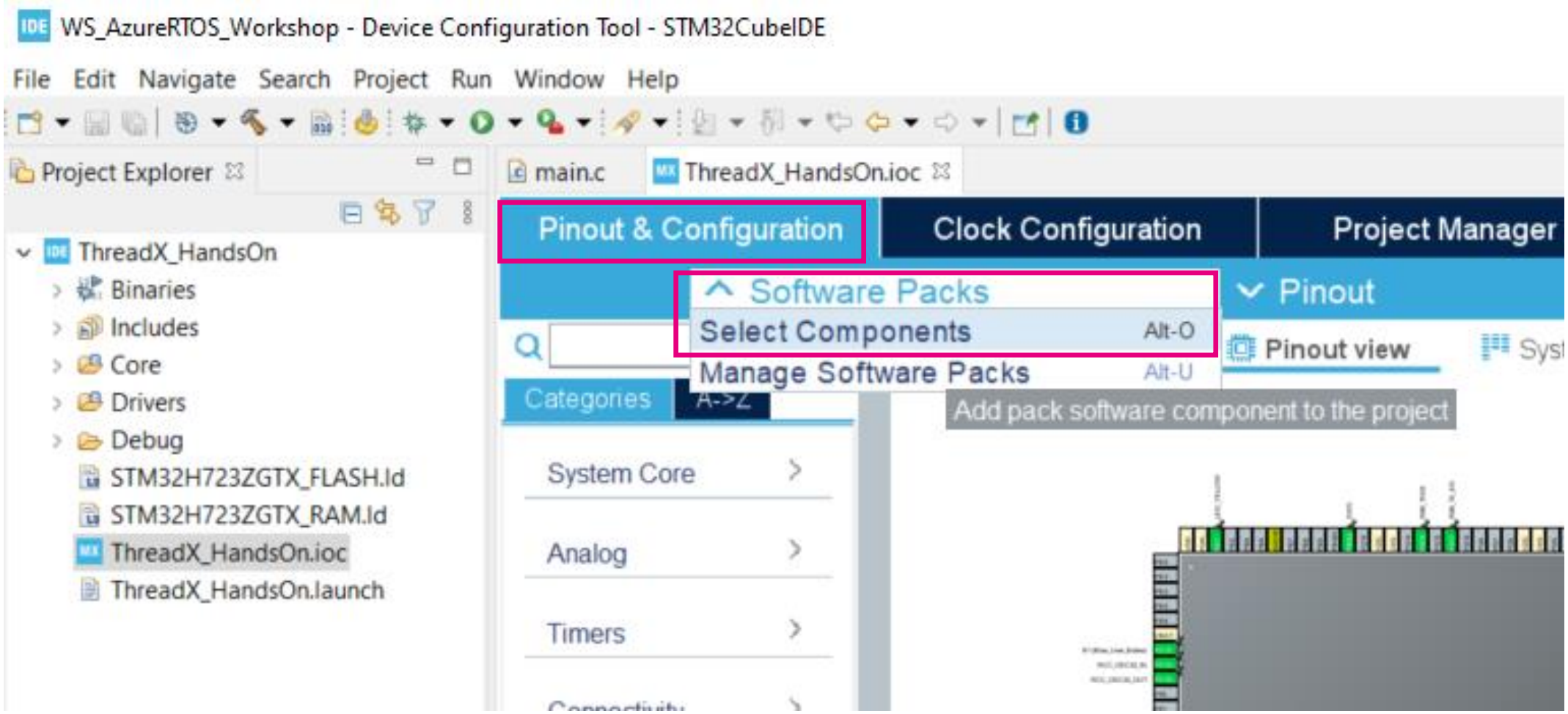
Switch to embedded CubeMX

- Terminate debug from first hands-on (CTRL-F2)
- Double click or right click/open on the file: ThreadX_HandsOn.ioc

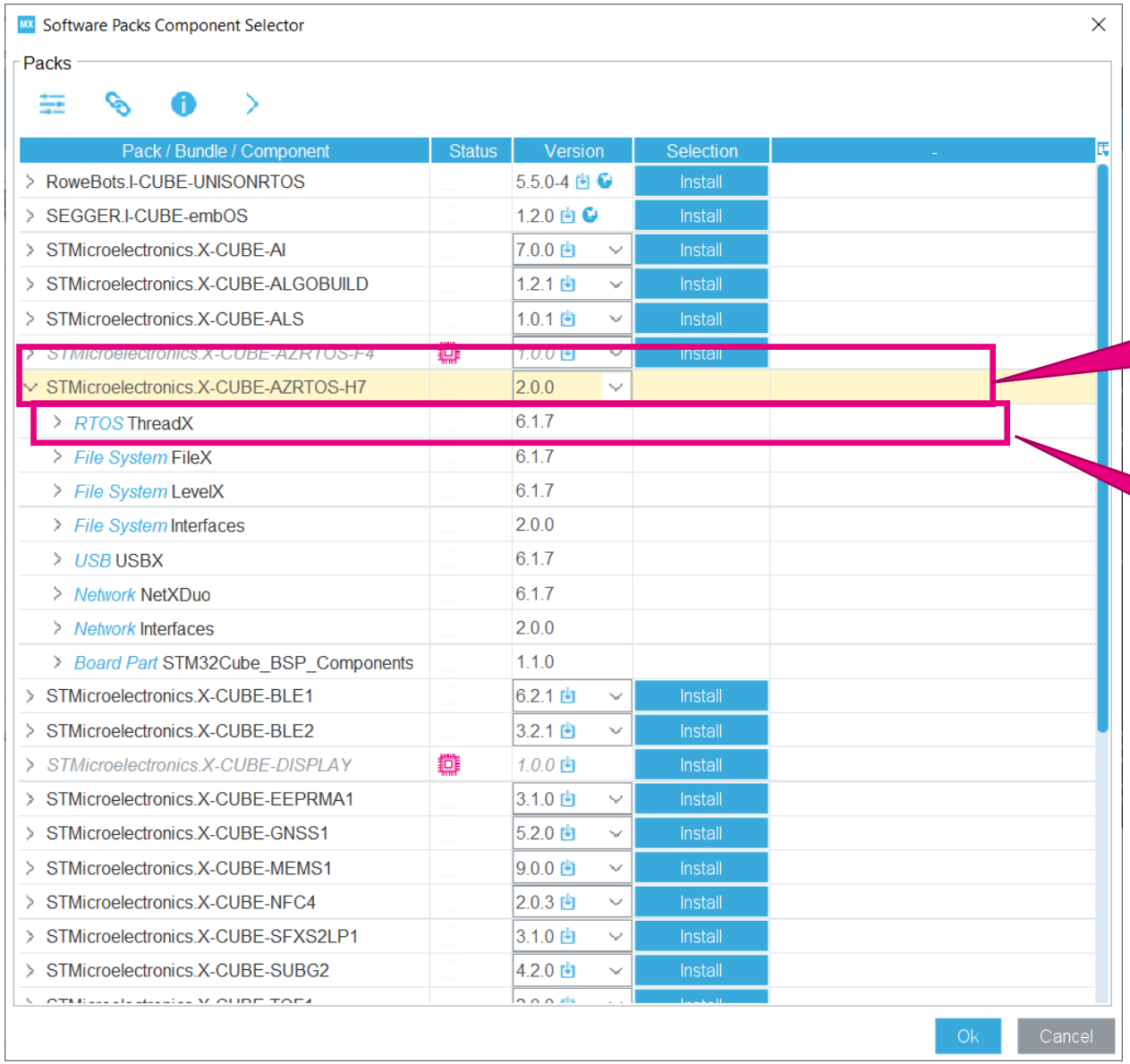


Select a software pack component

- Click on Software Pack and Select Components:



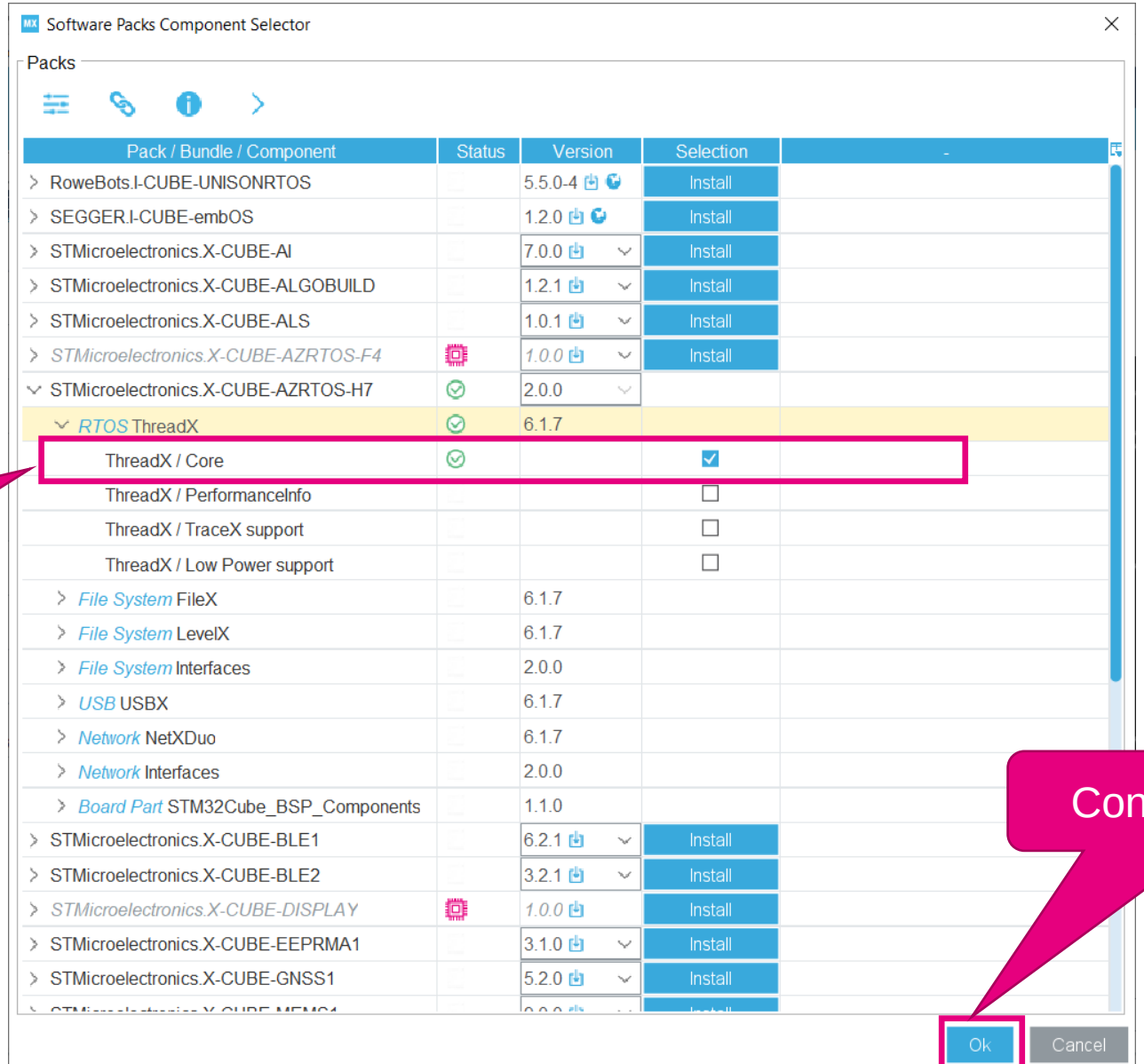
Component selection window



Open the X-CUBE-AZRTOS-H7

Open RTOS_ThreadX

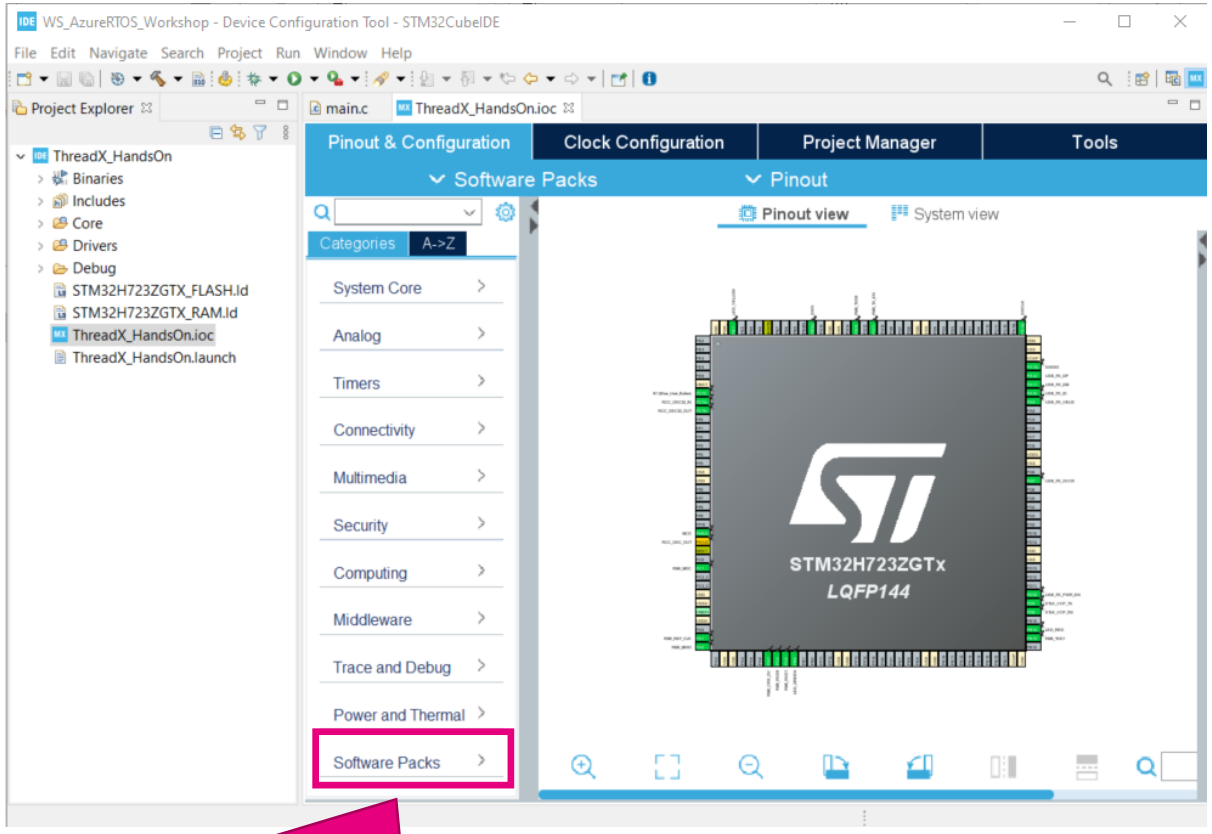
Component selection



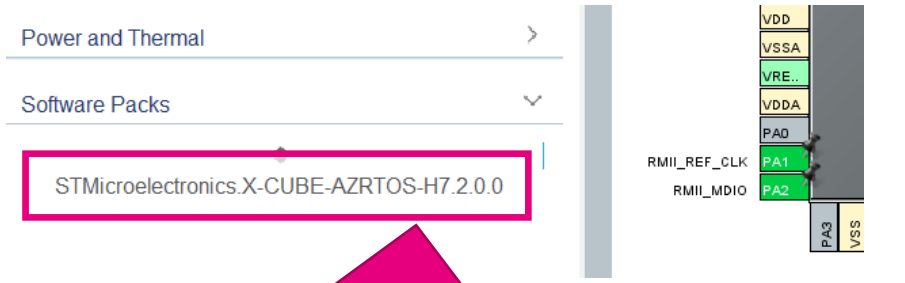
Select:
Application - azure_rtos_app

Confirm selection

Software pack included in the interface



New category added, open it

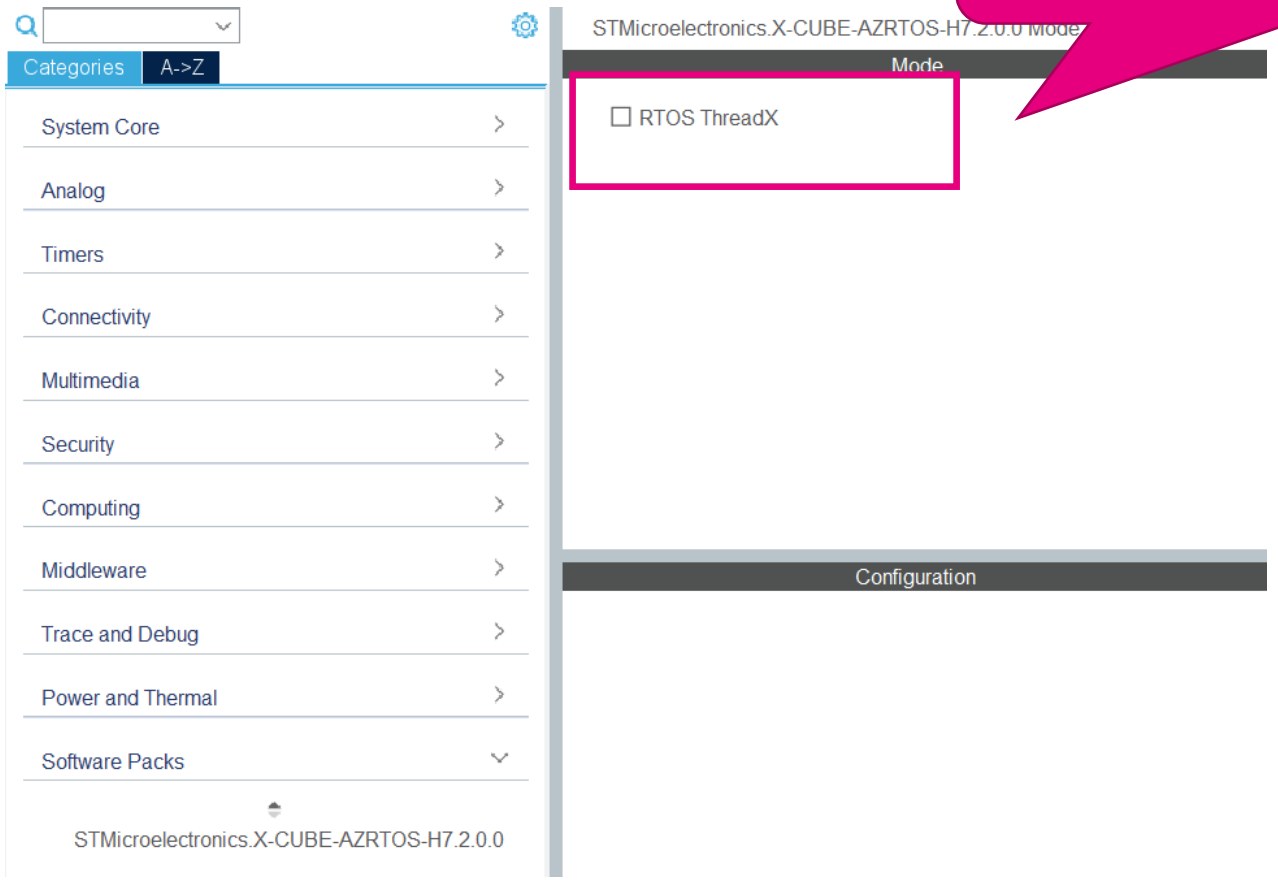


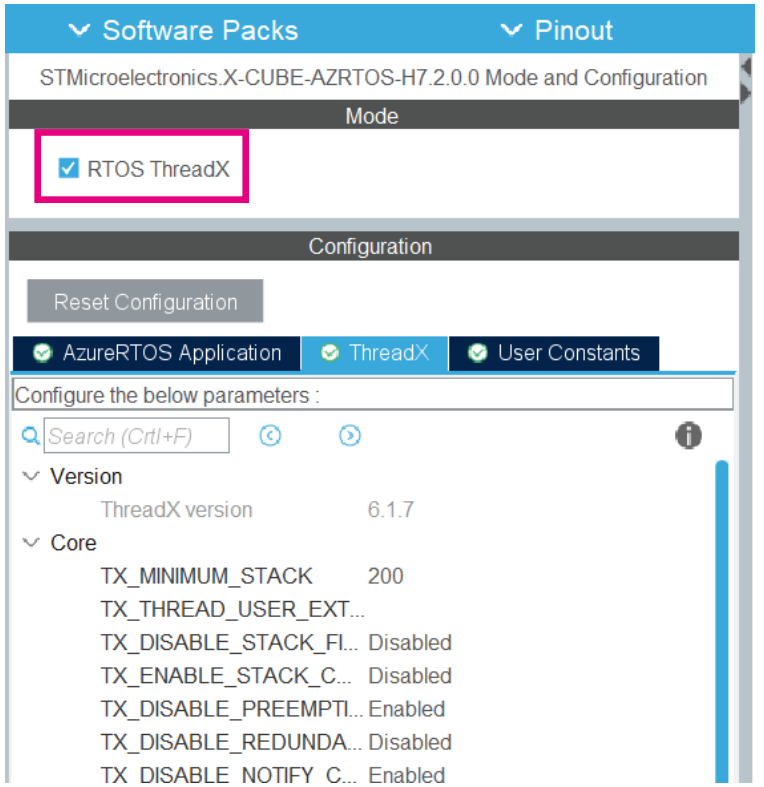
Select X-CUBE-AZRTOS_H7.2.0.0

life.augmented

Component selection in software pack

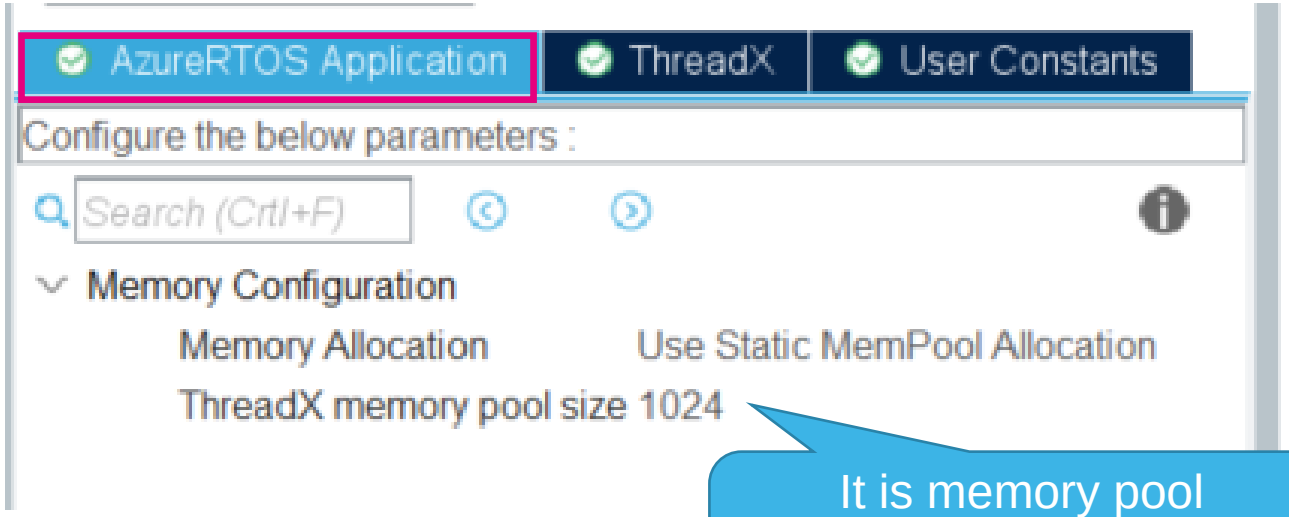
Select:
RTOS ThreadX



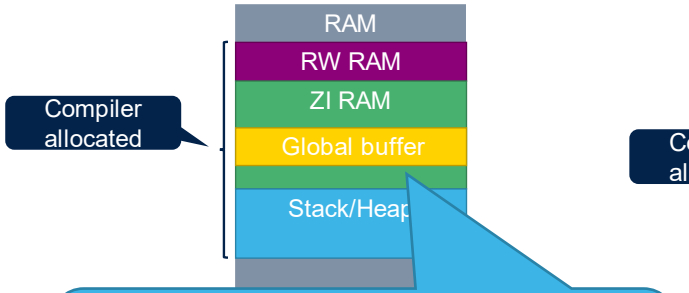


life.augmented

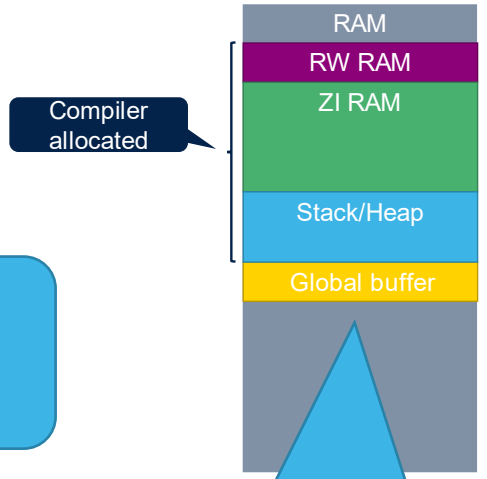
Azure RTOS application configuration TAB



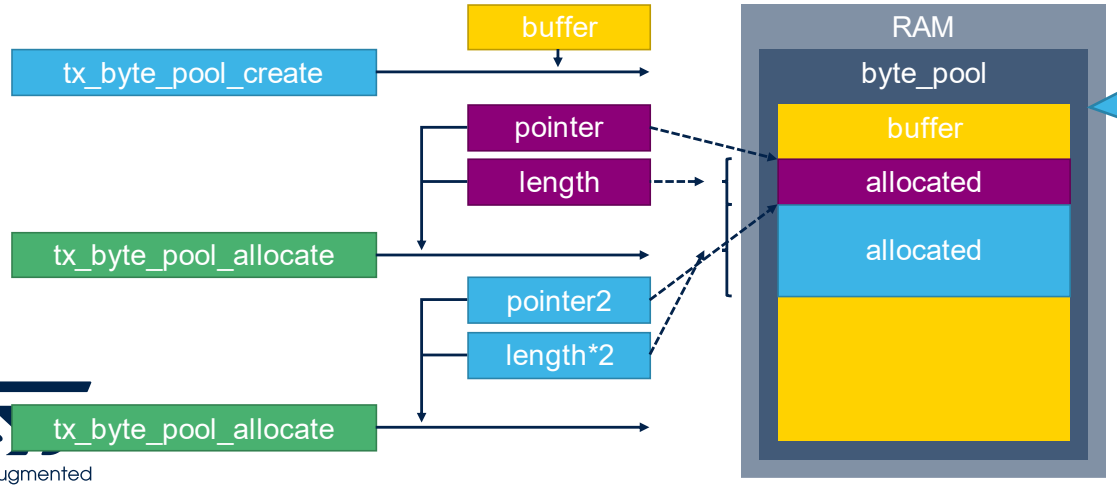
It is memory pool
Can be used to allocate memory by application



Static pool is allocated inside application space



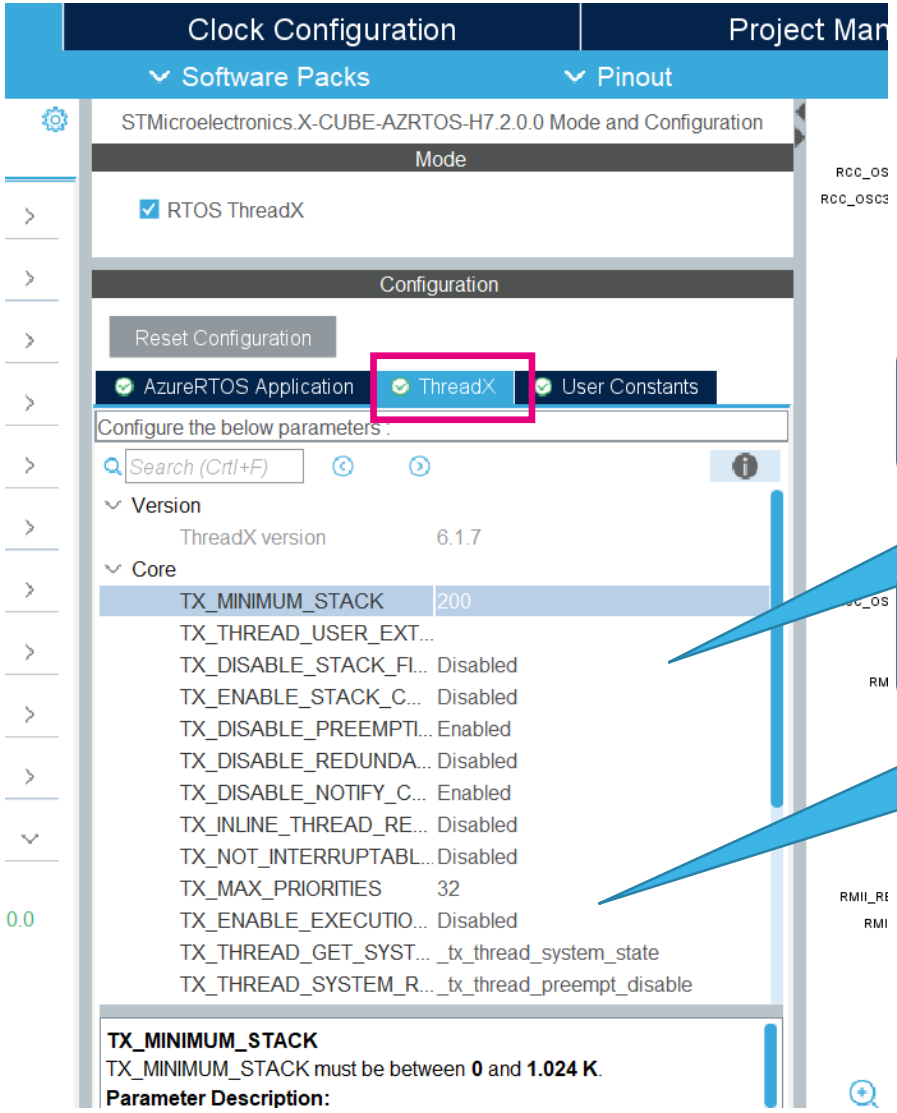
Dynamic pool is created outside application space
Last used RAM space



Application can obtain memory by calling tx_byte_pool_alloate

Using this mempool is optional

ThreadX configuration



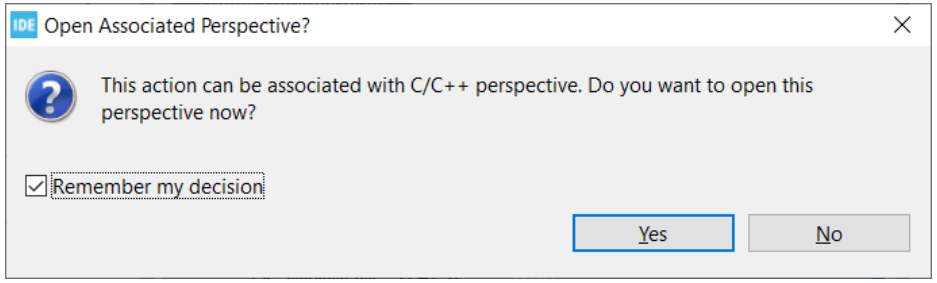
Can enable features of ThreadX

Fewer enabled feature will result in smaller ThreadX footprint

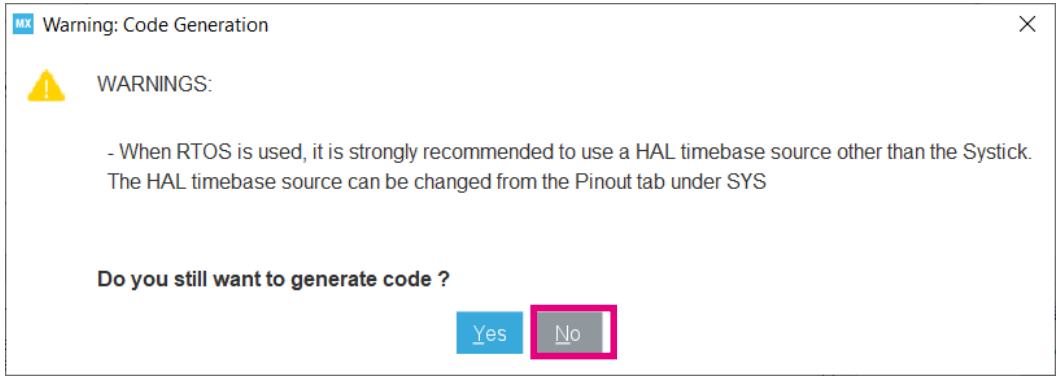
life.augmented

Generate new code structure

- Type ALT-K or Project/Generate Code
- You can request automatic change of perspective to C/C++

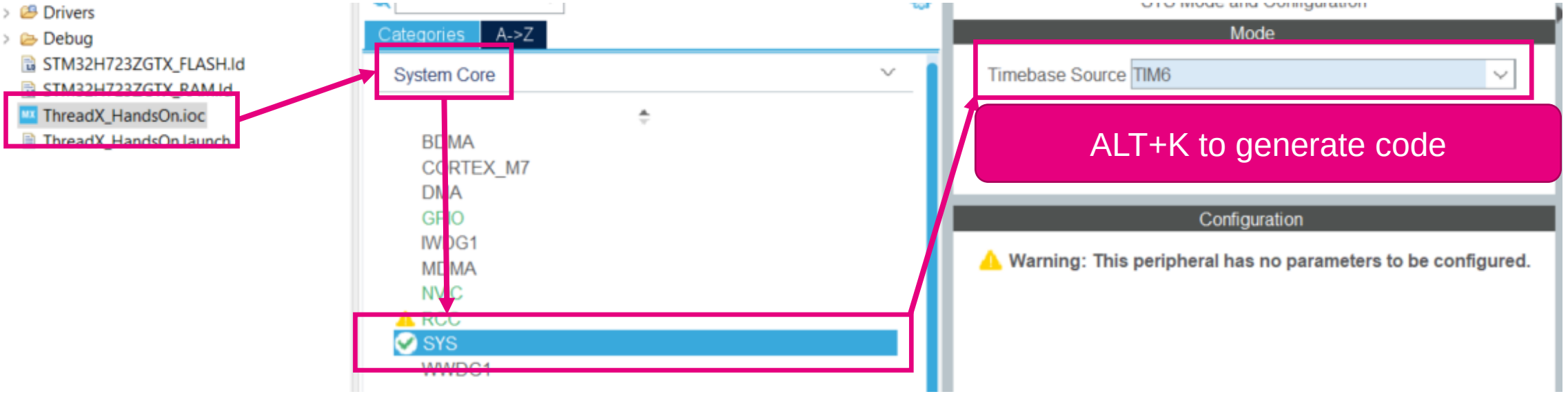


- But you will get a warning regarding the time base
- Please answer “No” and let’s address this point



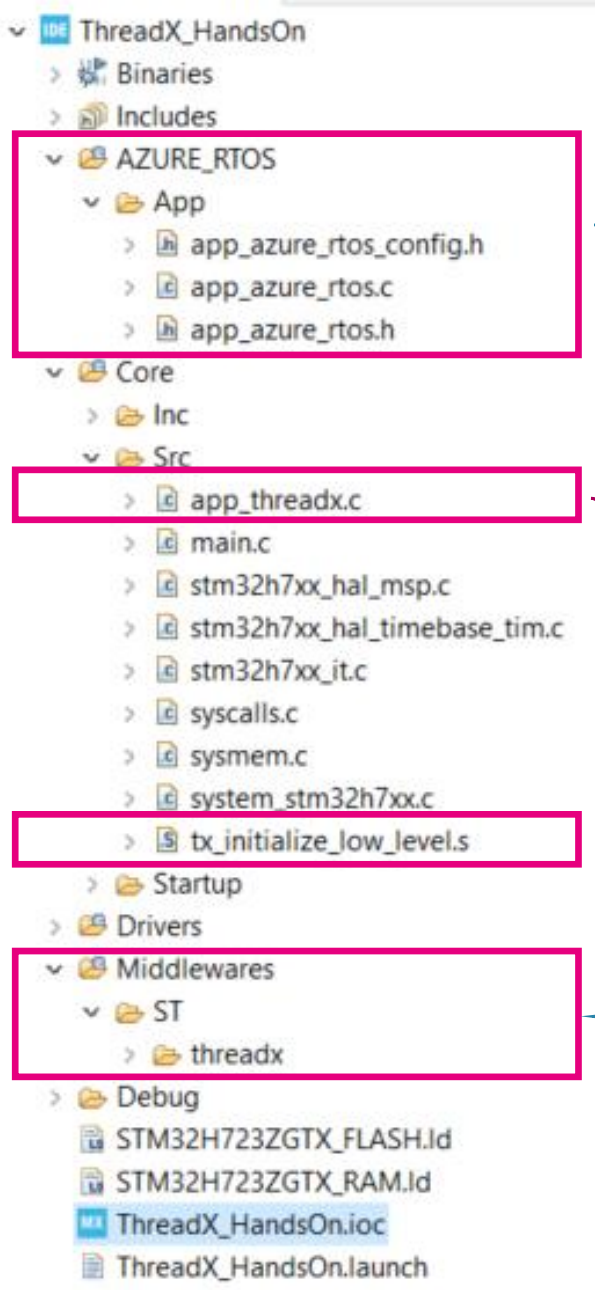
Selection of TimeBase

Timebase for HAL library



If HAL library don’t have separated time source the compilation will fail because both libraries want to use SysTick_Handler interrupt

Code generation



AZURE_RTOS\App contains the adaptation layer and configuration

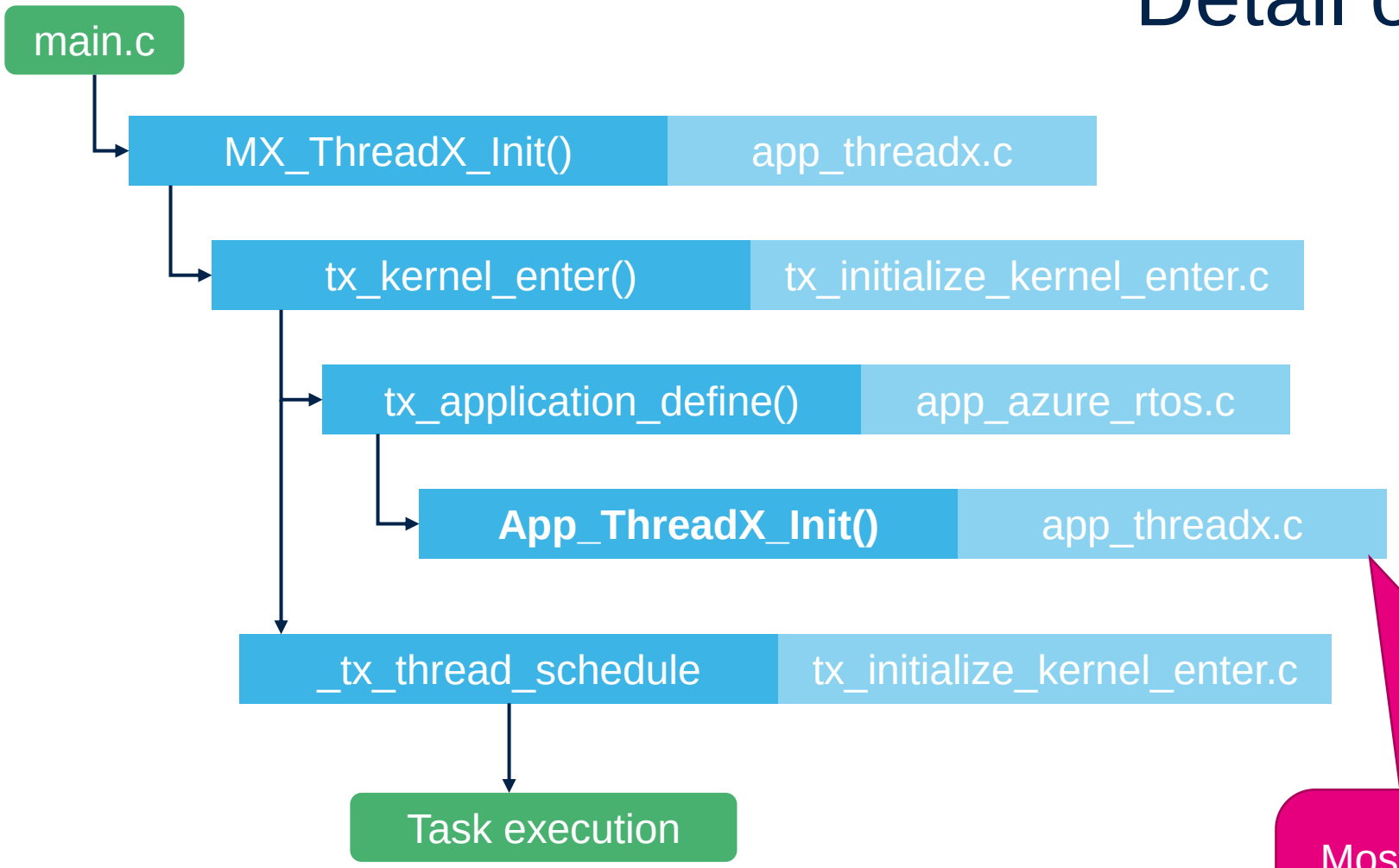
app_threadx.c place for user to create threads

ThreadX low level interface to STM32

Middlewares\ST\threadx contains all threadx source code

life.augmented

Detail on code structure



Most important is App_ThreadX_Init where user should create his first threads

Core\Src \app_threadx.c

Creating first thread : variables declarations

- Open Core\Src \app_threadx.c and add
  - To line 36 `/* USER CODE BEGIN PD */`
    - `#define THREAD_STACK_SIZE 1024`

```
35 /* Private define ---------------------
36  /* USER CODE BEGIN PD */
37  #define THRAD_STACK_SIZE 1024
38  /* USER CODE END PD */
39
```

Our stack size

  - To line 47 just after `/* USER CODE BEGIN PV */`

```
uint8_t thread_stack[THREAD_STACK_SIZE];
TX_THREAD thread_ptr;
```

```
45 /* Private variables ------------
46  /* USER CODE BEGIN PV */
47  uint8_t thread_stack[THRAD_STACK_SIZE];
48  TX_THREAD thread_ptr;
49  /* USER CODE END PV */
```

Our stack

Thread handle

  - To line 52 after `/* USER CODE BEGIN PFP */`

```
VOID my_thread_entry(ULONG initial_input);
```

```
51 /* Private function prototypes -------------
52  /* USER CODE BEGIN PFP */
53  VOID my_thread_entry(ULONG initial_input);
54  /* USER CODE END PFP */
```

Thread function prototype

Core\Src \app_threadx.c

STEP 2: Create the thread

- After the stack is allocated, we can create the thread: line 70
  - You can use completion with CTRL-Space at any time

```
tx_thread_create(
        &thread_ptr,
        "my_thread",
        my_thread_entry,
        0x1234,
        thread_stack,
        THREAD_STACK_SIZE,
        15,
        15,
        1,
        TX_AUTO_START
);
```

Thread handle defined from line 48

Name of thread

Thread function from line 53

Thread function input argument (any value you want 0-4G)

Thread stack from line 47

Thread stack size from line 37

Thread priority

Thread preemption priority

Thread time slice

Start of thread

```
  /* USER CODE BEGIN App_ThreadX_Init */
 tx_thread_create( &thread_ptr, "my_thread", my_thread_entry, 0x1234, thread_stack,
THREAD_STACK_SIZE, 15, 15, 1, TX_AUTO_START);
  /* USER CODE END App_ThreadX_Init */
```

life.augmented

Core\Src \app_threadx.c

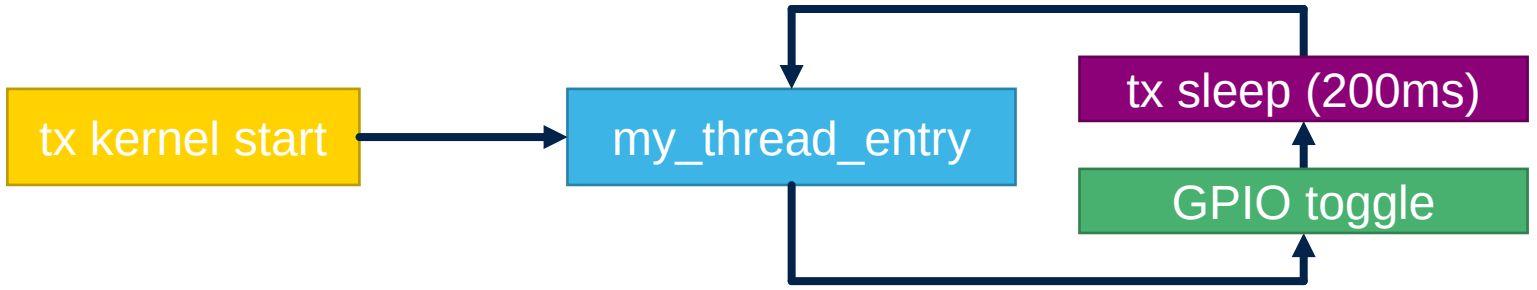
STEP 3: Create the thread's main function

- First, as we want to access to LED definitions, we need to include main.h
  - At line **26**, just after `/* USER CODE BEGIN Includes */`
    - `#include "main.h"`
  - Then after line **95**, `/* USER CODE BEGIN 1 */`

```
25 /* Private includes ----------
26 /* USER CODE BEGIN Includes */
27 #include "main.h"
28 /* USER CODE END Includes */
29
```

```
95 /* USER CODE BEGIN 1 */
96 VOID my_thread_entry (ULONG initial_input)
97 {
98     while(1){
99         HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
100        tx_thread_sleep(20);
101    }
102 }
103 /* USER CODE END 1 */
```

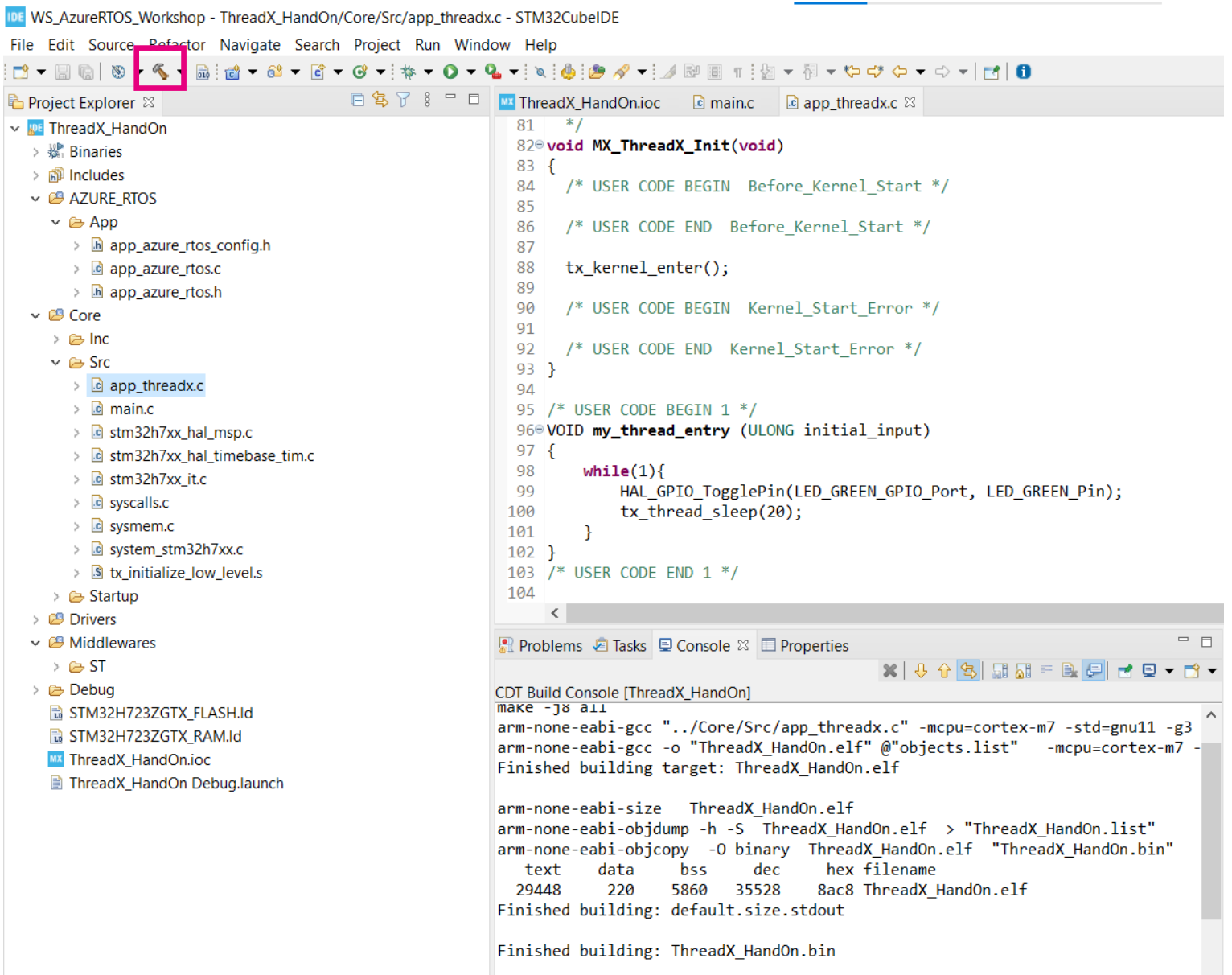
```
VOID my_thread_entry (ULONG initial_input)
{
    while(1){
        HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
         tx_thread_sleep(20);
    }
}
```



life.augmented

Build the project

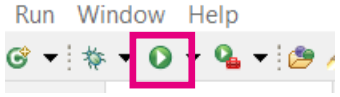
- Use hammer icon or Project/Build Project



life.augmented

Flash the new thread project

- The configuration from bare metal can be reused to flash the board.
- You can use either
  - the green arrow button
  - Or Run/Run

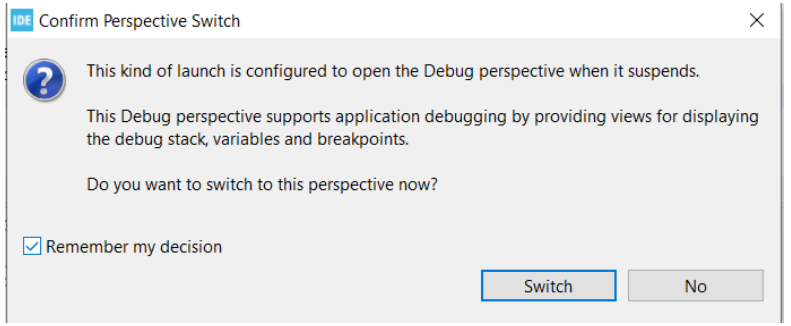


- You should get now a LED blinking with 400 ms period

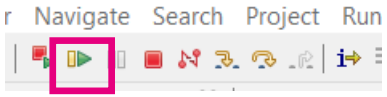
Start the debugger

- Click on debugger icon :

- Pop-up window to switch perspective
  - Click remember my decision
  - Click on switch to activate the debug perspective

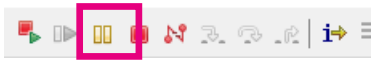


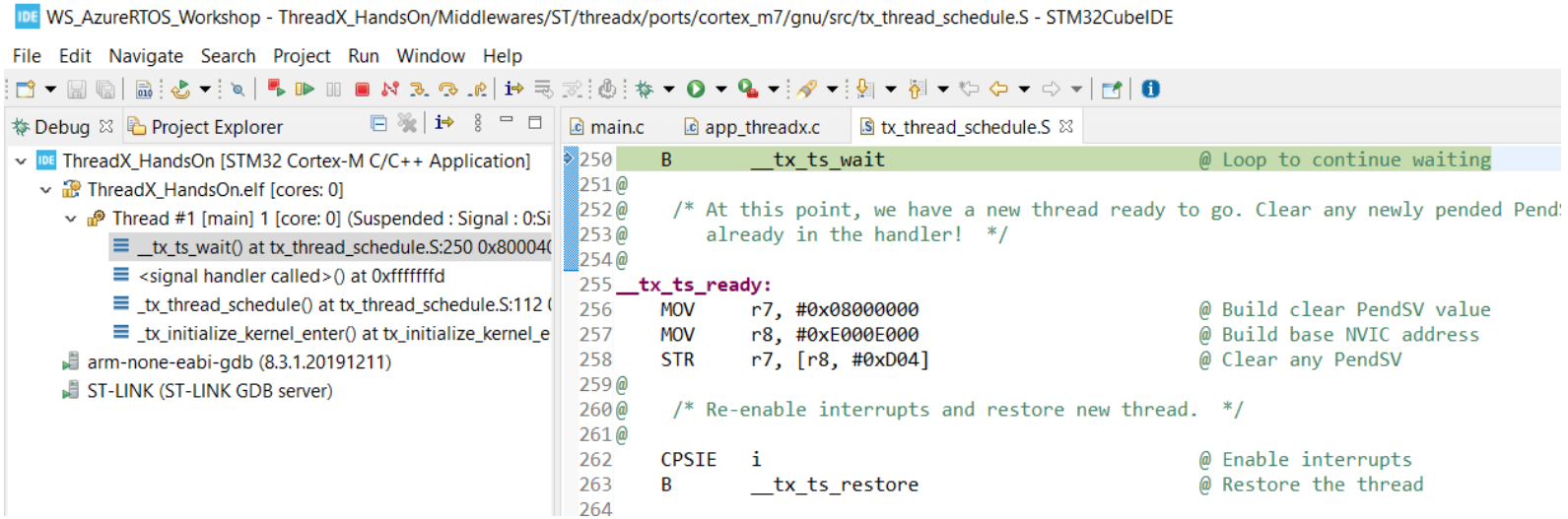
- Run the code by clicking either
  - Resume icon
  - F8
  - Run/Resume



- Green led should blink

Suspend the execution

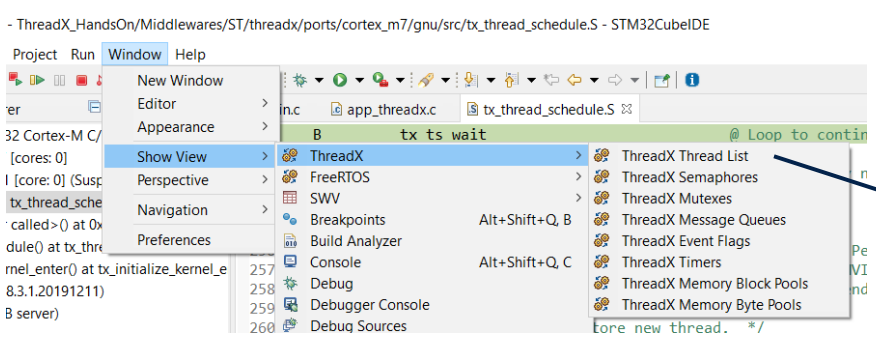
- Break the running firmware using either
  - Break icon 
  - Run/Suspend
- Suspended in the ThreadX scheduler

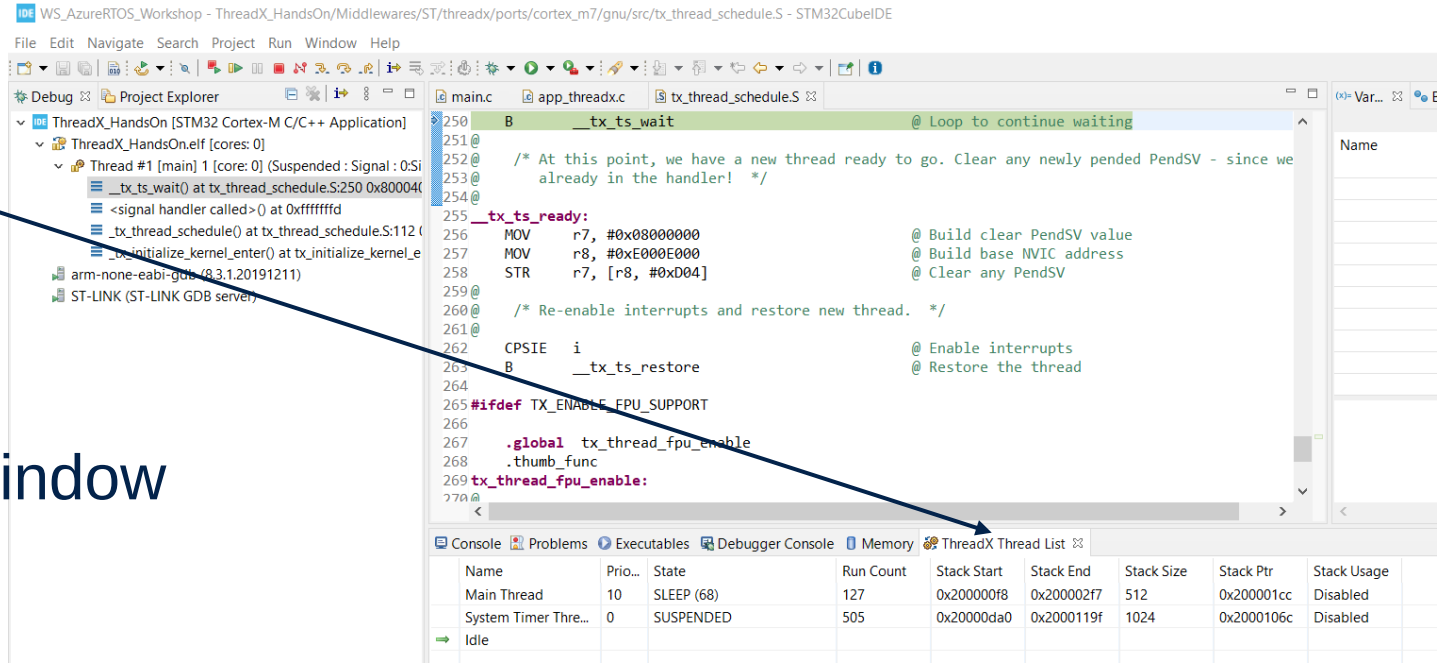


life.augmented

ThreadX aware debugging

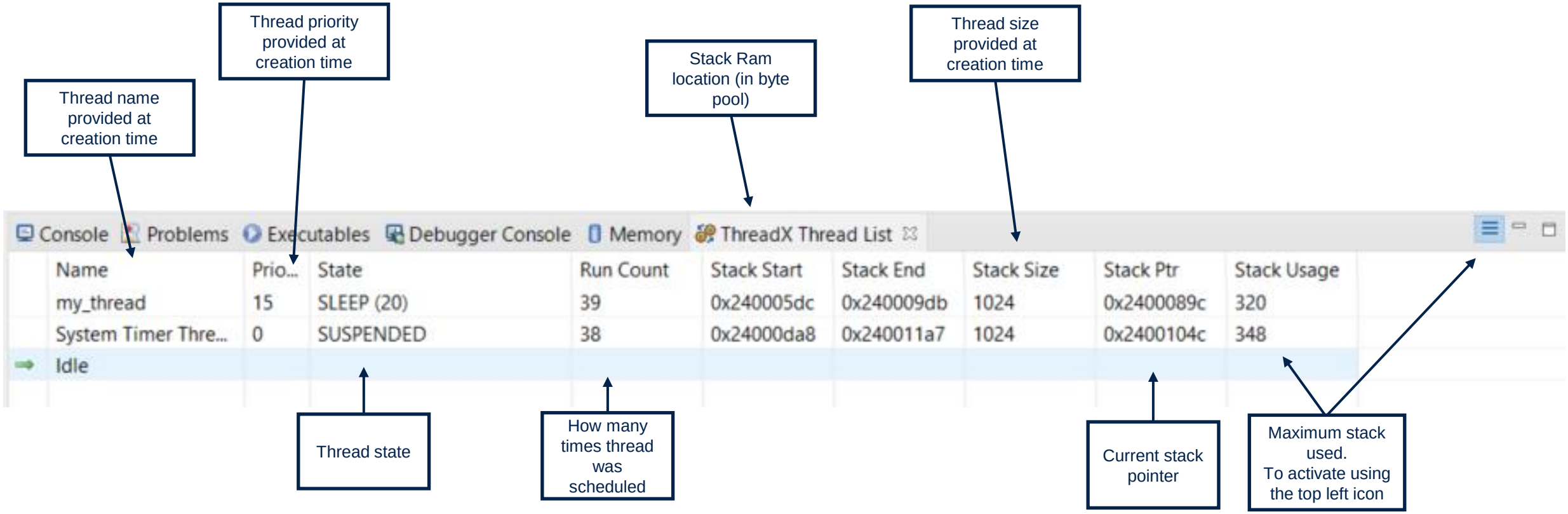
- Threads can be viewed using specific view
- Please open the ThreadX thread list view
  - Window/Show View/ThreadX/ThreadX Thread List





- It opens a new view in bottom window

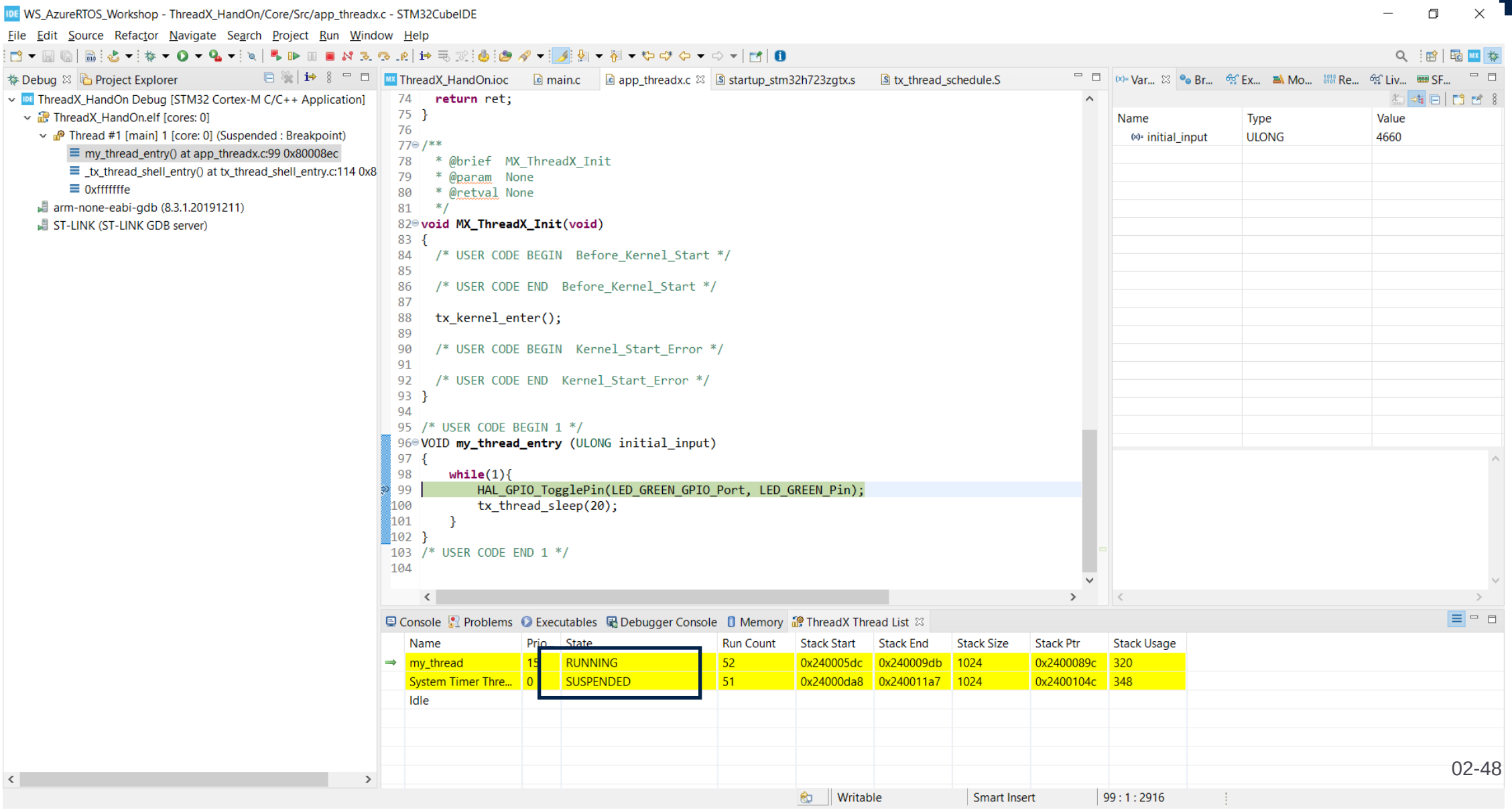
ThreadX debug view



Add a breakpoint to see impact on thread list

- Resume execution
- Open app_threadx.c and go to end of file
- Click on the line containing the HAL_GPIO_TogglePin
- Set a breakpoint here with either methods
  - Double click on the column just on the left of the line numbers
  - CTRL-SHIFT-B
- As soon as the breakpoint is created, the execution will stop with maximum 200 ms delay
- The ThreadX thread view is now updated …

Updated ThreadX thread list



life.augmented

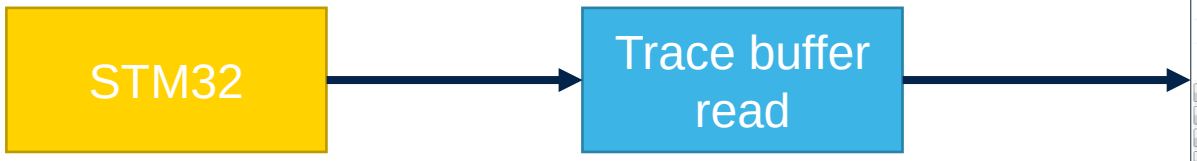
Conclusion

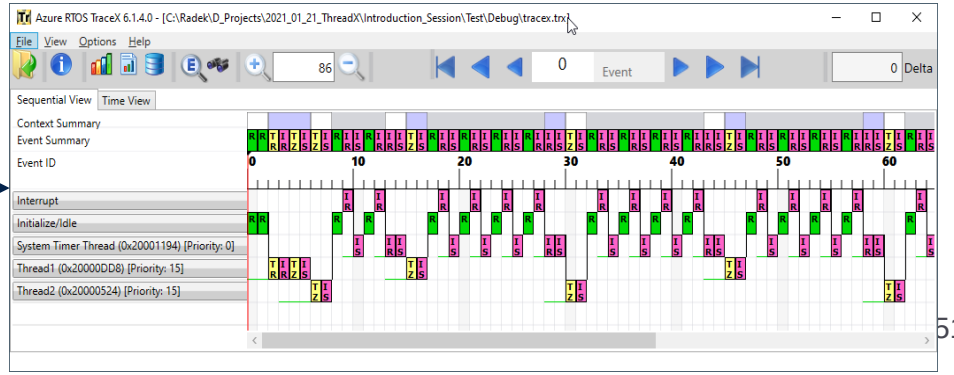
- Adding ThreadX application to a bare metal project consists in
  - Adding the ThreadX component in STM32CubeMX
  - Adapt ThreadX configuration in STM32CubeMX
  - Generate the code from STM32CubeMX
    - This will update the project structure with all needed files
  - Use ThreadX API to create ThreadX components thread, mutex, etc
- This is the integration of ThreadX in our Cube ecosystem

Step 3: Activate TraceX

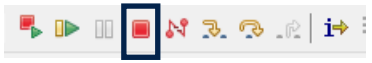
Purpose

- We will see in the new hands-on how to activate TraceX feature of ThreadX
- TraceX allows a post mortem analysis of the thread sequence occurred during firmware execution
- To make this work we need to
  - activate the feature in the firmware
  - allocate memory to store the trace during firmware execution
  - execute the firmware to fill the trace in memory
  - write the content of the memory in a file on the PC
  - use Microsoft's TraceX tool to display the result





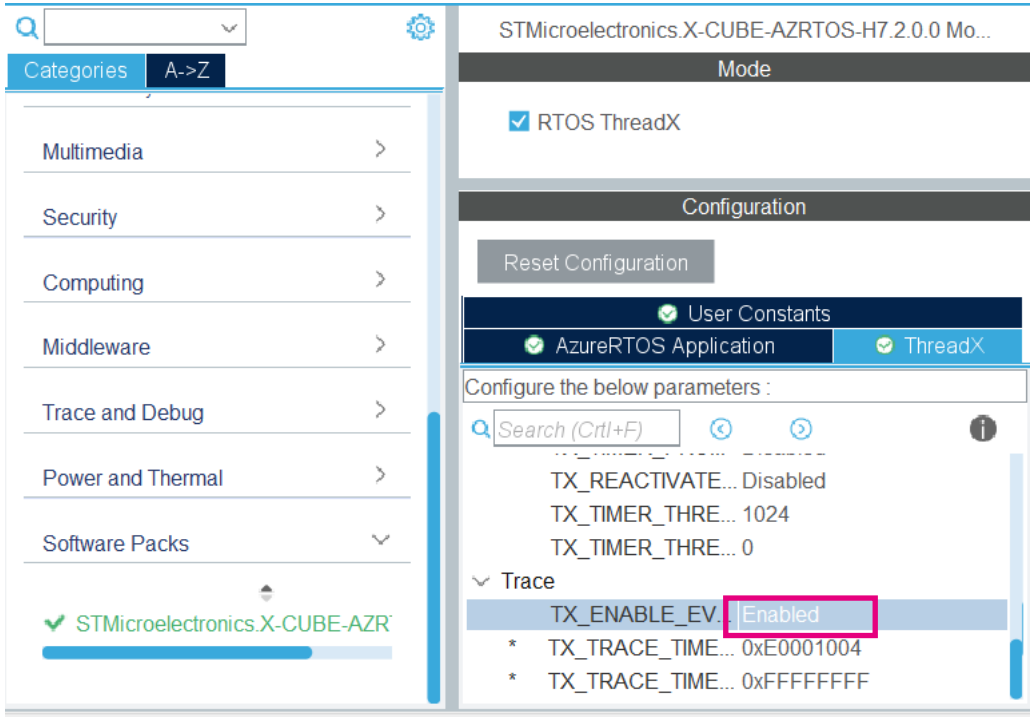
Activating traceX

- First close the debugger from previous hands-on
  - Click on the red square 
  - Run/Terminate
  - CTRL-F2
- Activate embedded STM32CubeMX: double click on ThreadX_HandsOn.ioc
- Go to the Software Packs/ Select component or ALT-O
- Open STMicroelectronics.X-CUBE-AZRTOS-H7/RTOS ThreadX
- Activate TraceX support
- OK to validate

life.augmented

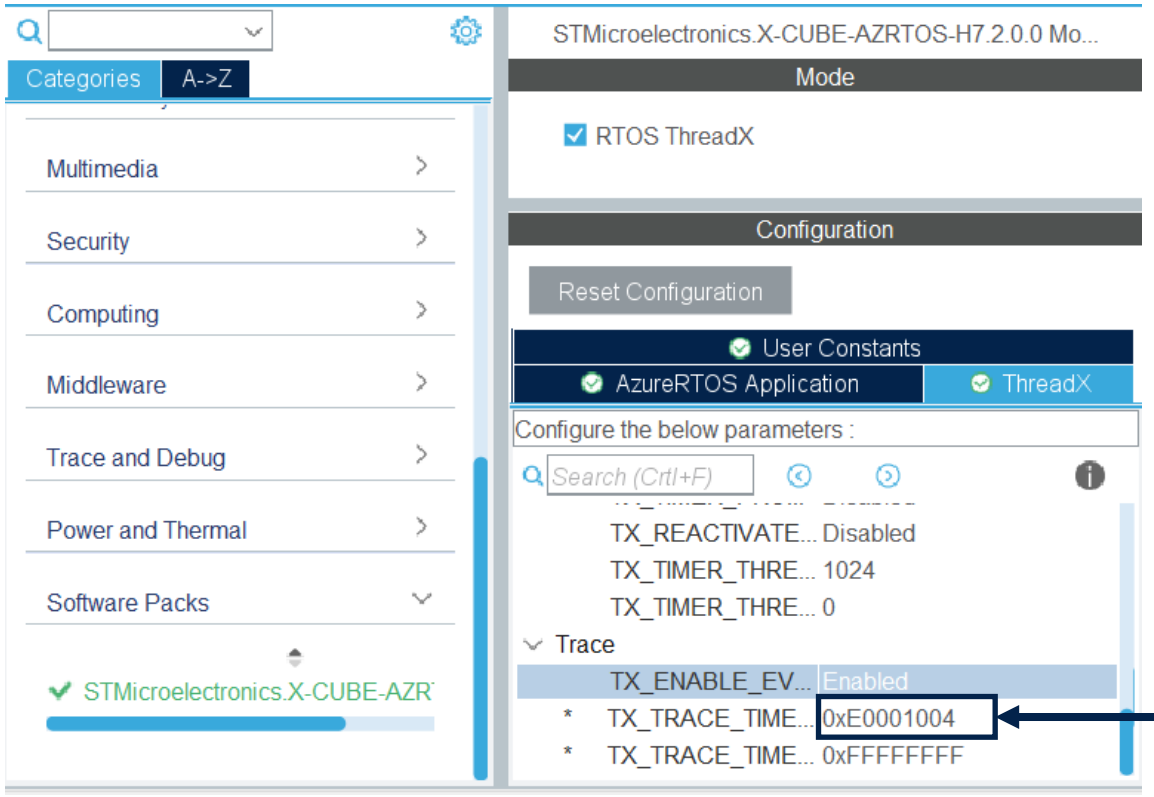
Enable the traces

- In left panel of CubeMX select Software pack
- In middle panel select ThreadX tab
- Change value of configuration TX_ENABLE_EVENT_TRACE to Enabled



Trace timestamp source

- The trace timestamp source is setup by default on the cycle counter of the Cortex at address 0xE0001004



Use ALT-K to update the generated code for TraceX support

Core\Src \app_threadx.c

Memory dedicated to trace

- In app_threadx.c into section `/* USER CODE BEGIN PV */` Line **47** add

```
#define TRACEX_BUFFER_SIZE 64000
uint8_t tracex_buffer[64000] __attribute__ ((section (".trace")));
```

put buffer to specific RAM section

Buffer used for trace

```
49
50 #define TRACEX_BUFFER_SIZE 64000
51 uint8_t tracex_buffer[64000] __attribute__ ((section (".trace")));
52 /* USER CODE END PV */
```

- To line **73** add:

```
tx_trace_enable(&tracex_buffer,TRACEX_BUFFER_SIZE,30);
```

Amount trace objects in buffer

Our trace buffer

buffer size

```
74
75   tx_trace_enable(&tracex_buffer,TRACEX_BUFFER_SIZE,30);
76   /* USER CODE END App_ThreadX_Init */
77
```

life.augmented

STM32H723ZGTX_FLASH.ld

Put trace buffer to AXI SRAM

- Add this to STM32H723ZGTX_FLASH.ld file around line 120

```
.trace (NOLOAD):
{
 . = ALIGN(4);
    *(.trace)
} > RAM_D1
```

Put .trace section to AXI SRAM

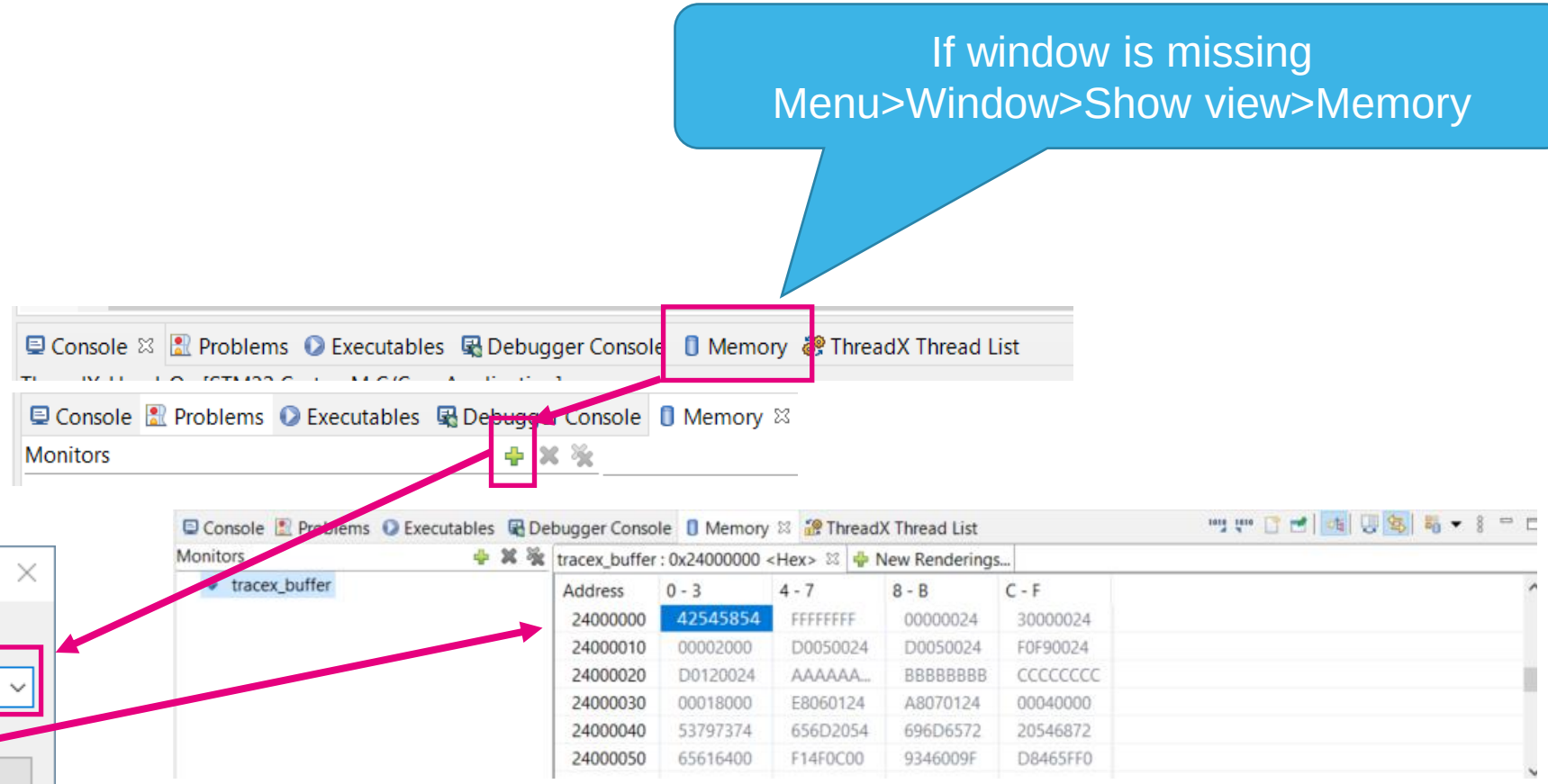
```
120  } >FLASH
121
122 .trace (NOLOAD):
123 {
124  . = ALIGN(4);
125     *(.trace)
126 } > RAM_D1
127
128  /* used by the startup to initialize data */
129  _sidata = LOADADDR(.data);
130
```

Now the trace buffer will be on known location 0x24000000

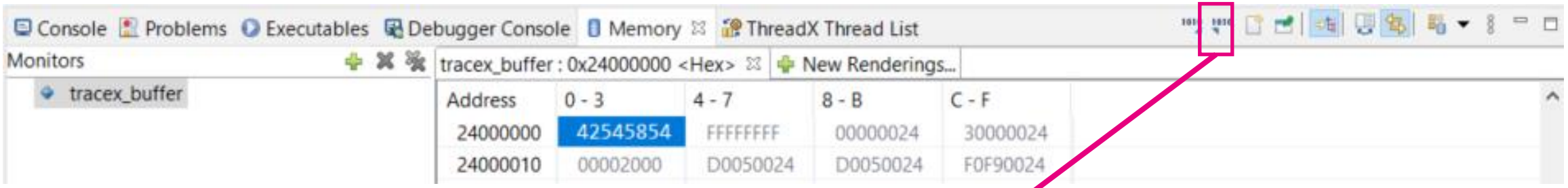
life.augmented

Execute firmware in debugger

- Fill the trace buffer by running project
  - Compile project
  - Run debug
  - Run code
  - Suspend debug
- Read trace buffer
  - Open memory window
  - Add new memory area
  - Type tracex_buffer



TraceBuffer export

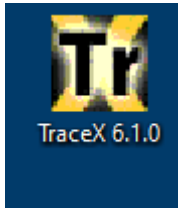


- Please click on export icon
- A pop-up should look like this
  - Select format **Raw binary**
  - Type length **64000**
  - Click on Browse and create a directory in tmp
    - for instance: c:\tmp\AZRTOSLogs
  - Then use filename log1.**trx**
  - Then press ok : the log is written

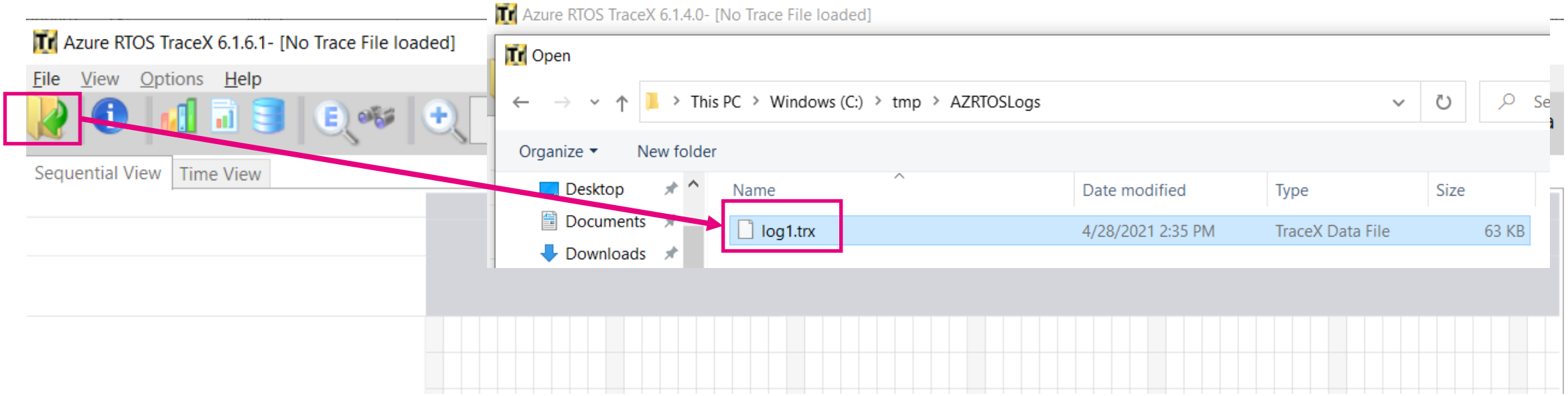
TRX extension is important for TraceX App

Open TraceX

- You should have TraceX installed. Open the application

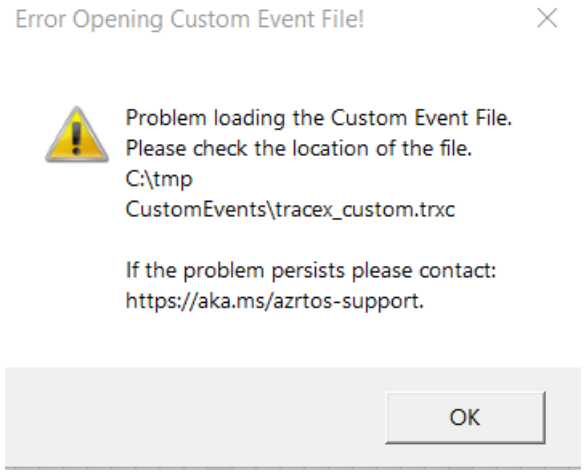


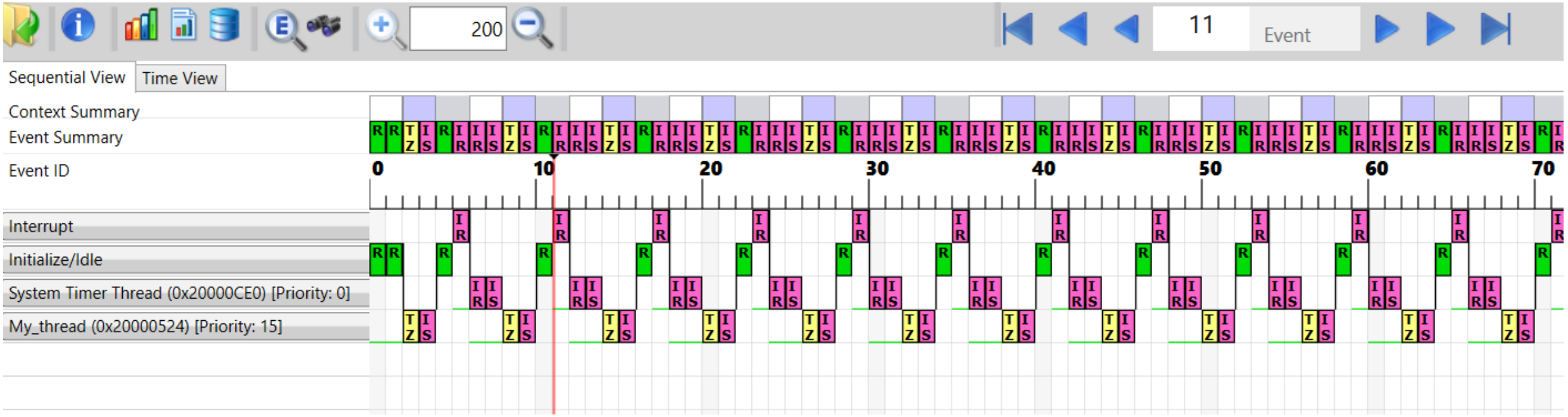
- Open file and select the log file just created



Show the trace

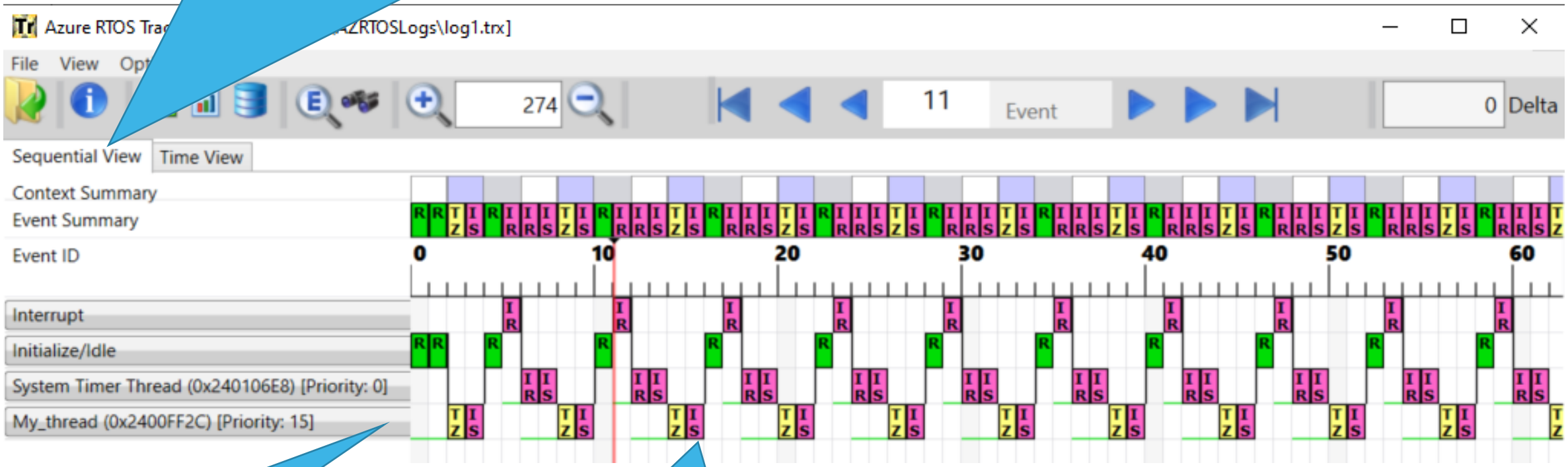
- An error pop-up should appear:
- Just press OK
- You get the trace





Analyzing the trace

The sequential view does not provide any timing information, just the scheduling sequence

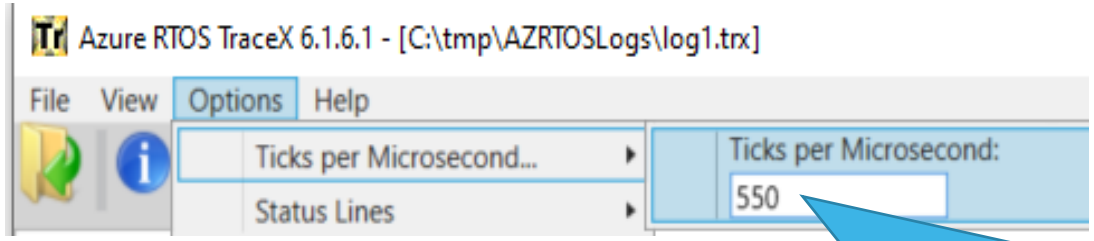


Our thread running GPIO toggle

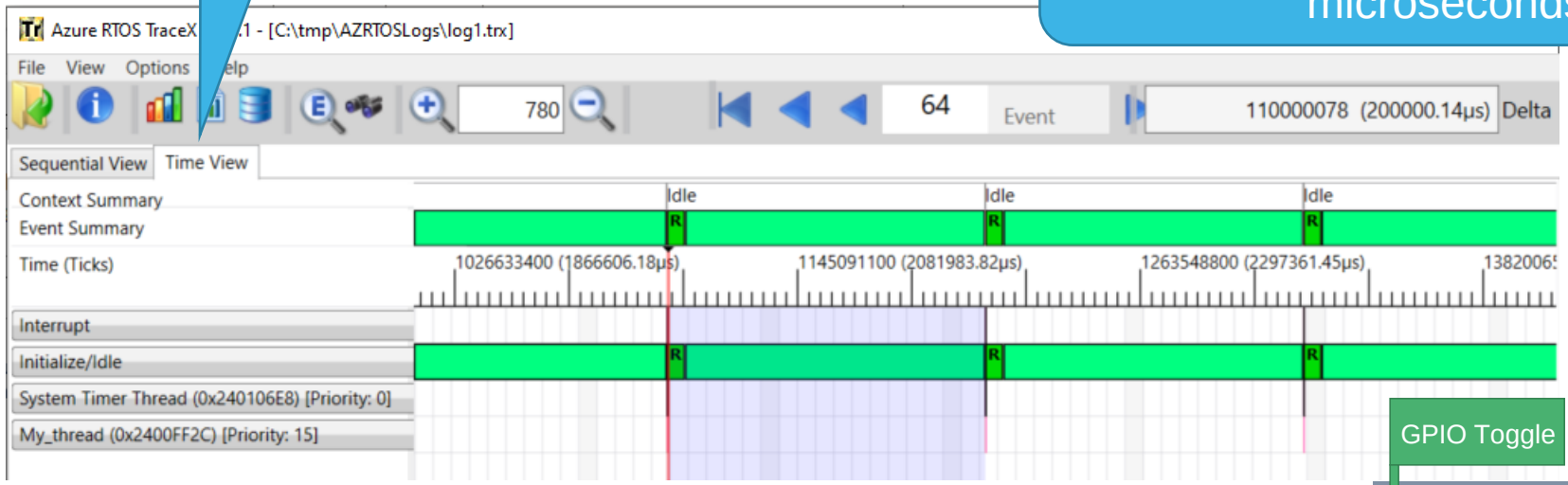
Thread is going to sleep

Using Time View

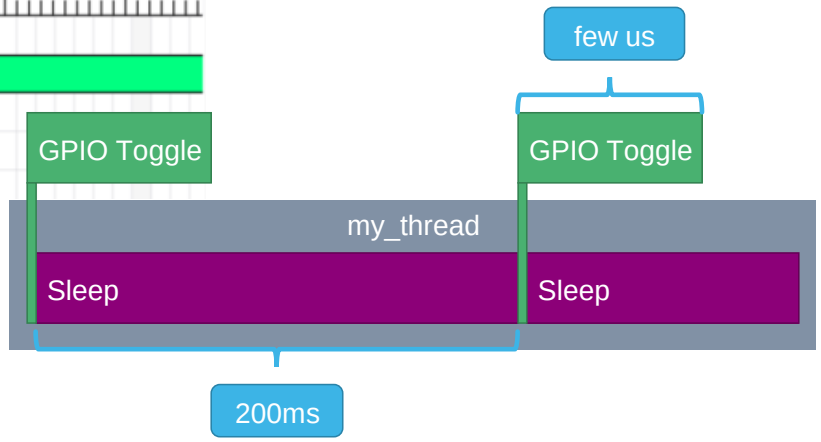
You can check the Time view



To be able to use it efficiently you need to provide the tick per microseconds in menu Options/Ticks per microseconds: Please provide 550



Our thread runs only few clock cycles before sleep



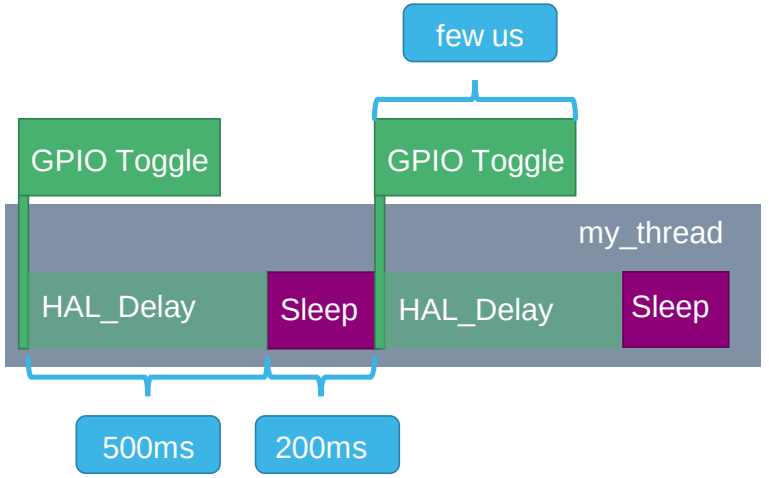
life.augmented

Change behaviour

- In order to show how TraceX can be used, we will add one change in the code
- This change consists in using HAL_Delay which is performing an active wait
- In MainThread add the following lines:

```
HAL_Delay(500);
```

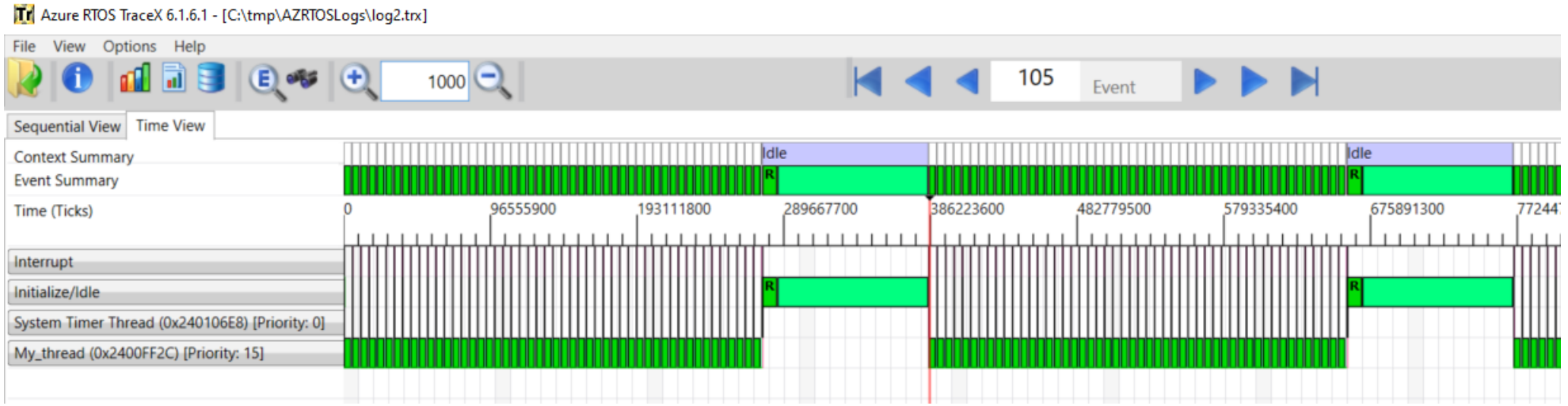
```
102 /* USER CODE BEGIN 1 */
103 VOID my_thread_entry (ULONG initial_input)
104 {
105     while(1){
106         HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
107         HAL_Delay(500);
108         tx_thread_sleep(20);
109     }
110 }
```



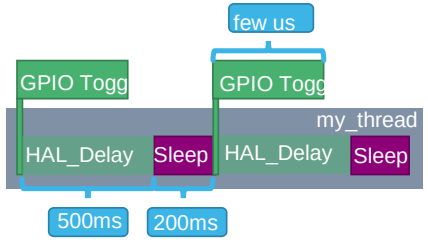
Obtain new trace

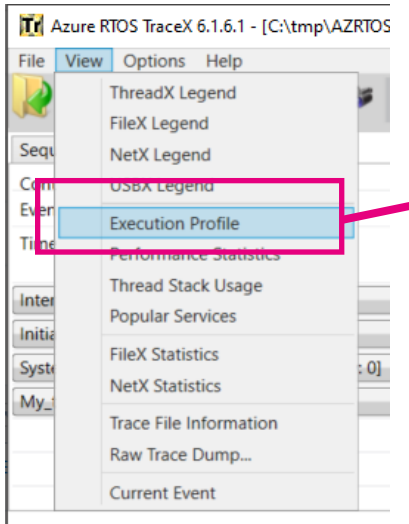
- You can launch directly the debugger. Recompilation will be done automatically.
- Launch the firmware : LED should blink with longer period
- Break after few seconds
- Export the trace buffer:
  - Click on memory. The TraceBuffer should be already displayed
  - Double click on Hex Integer to create a new rendering
  - Click on export button and replace log1.trx by log2.trx, then click OK
  - Go back to TraceX application, click on open file icon or File/Open
  - Default directory should be same as before, just select log2.trx
  - In the sequential view you shouldn’t see much difference
  - Select Time view and unzoom

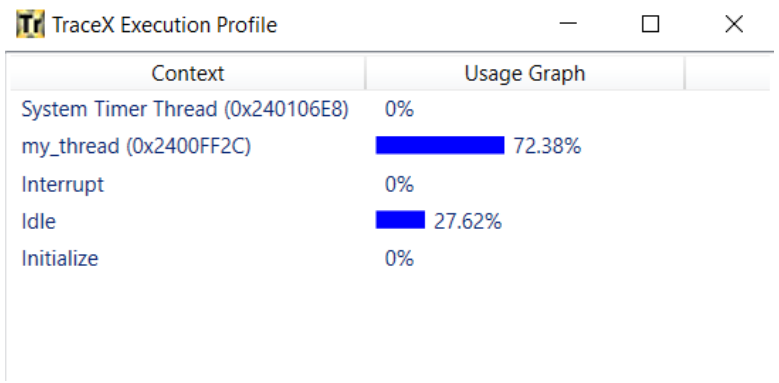
New trace



- In time view the my_thread active wait is immediately visible.
- Also, launching View/Execution profile
  - The my_thread time consumption is obvious!





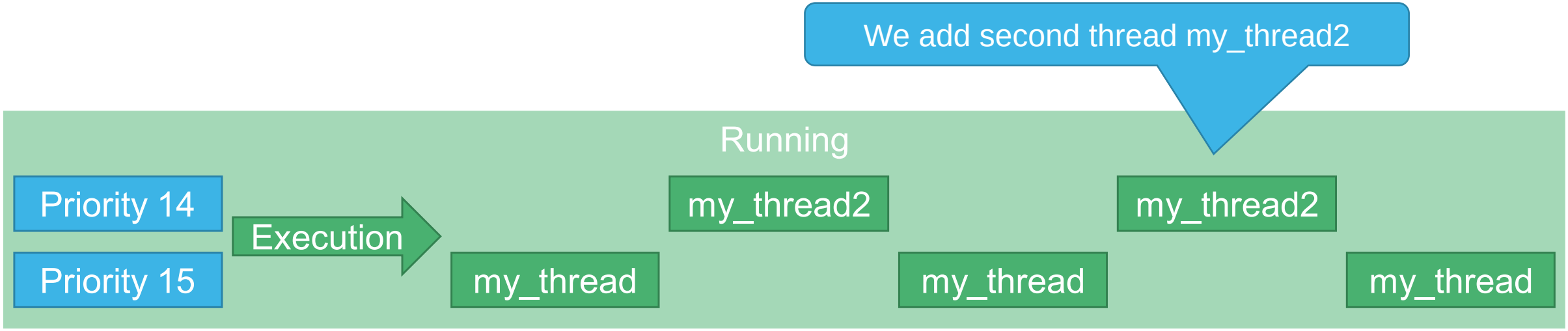


life.augmented

Step 4 : Concurrent threads

Purpose

- Now we have seen the ThreadX environment, we will show one specific feature of this RTOS that is preemption threshold
- Let’s have 2 concurrent threads and see how the scheduler manages the priorities



Add a second thread

Core\Src \app_threadx.c

- Edit App_threadx.c
  - Add new thread handle and stack

```
uint8_t thread_stack2[THREAD_STACK_SIZE];
TX_THREAD thread_ptr2;
```

  - Function prototype

```
VOID my_thread_entry2(ULONG initial_input);
```

  - Create thread priority 14

```
tx_thread_create(&thread_ptr2, "my_thread2", my_thread_entry2, 0x1234, thread_stack2, THREAD_STACK_SIZE, 14, 14, 1, TX_AUTO_START);
```

  - Thread function

```
VOID my_thread_entry2 (ULONG initial_input)
{
    while(1){
        HAL_GPIO_TogglePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin);
        HAL_Delay(500);
        tx_thread_sleep(20);
    }
}
```

```
46 /* USER CODE BEGIN PV */
47 uint8_t thread_stack[THREAD_STACK_SIZE];
48 TX_THREAD thread_ptr;
49 uint8_t thread_stack2[THREAD_STACK_SIZE];
50 TX_THREAD thread_ptr2;
51 #define TRACEX_BUFFER_SIZE 64000
52 uint8_t tracex_buffer[64000] __attribute__ ((section (".trace")));
53 /* USER CODE END PV */
```

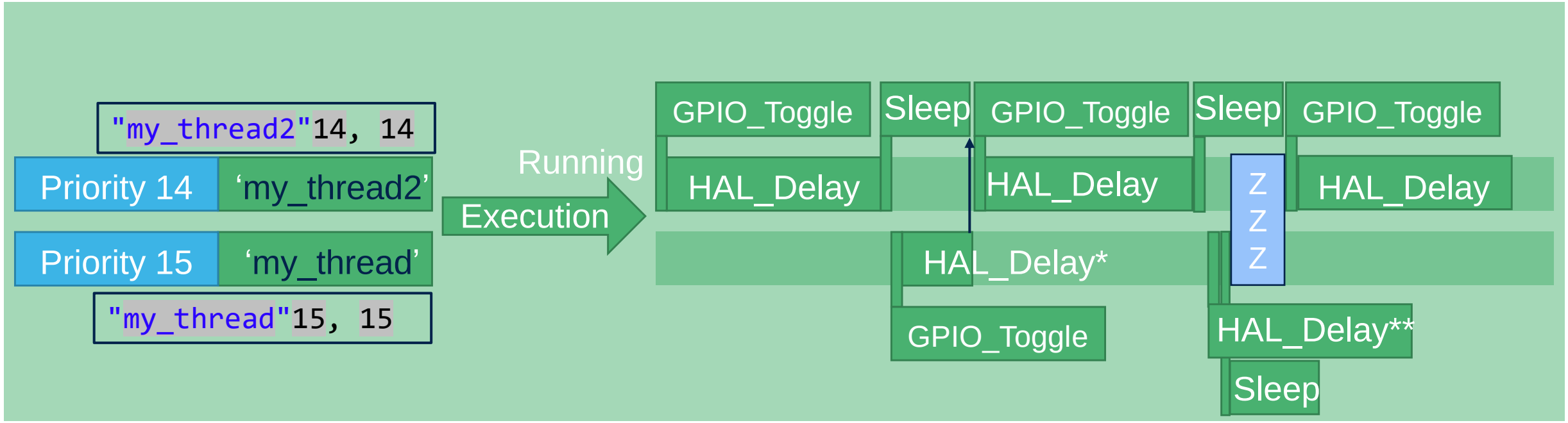
```
55 /* Private function prototypes -----------
56 /* USER CODE BEGIN PFP */
57 VOID my_thread_entry(ULONG initial_input);
58 VOID my_thread_entry2(ULONG initial_input);
59 /* USER CODE END PFP */
```

```
75 /* USER CODE BEGIN App_ThreadX_Init */
76 tx_thread_create( &thread_ptr, "my_thread", my_thread_entry, 0x1234, thread_stack, THREAD_STACK_SIZE, 15, 15, 1, TX_AUTO_START);
77 tx_thread_create(&thread_ptr2, "my_thread2", my_thread_entry2, 0x1234, thread_stack2, THREAD_STACK_SIZE, 14, 14, 1, TX_AUTO_START);
```

```
112 VOID my_thread_entry2 (ULONG initial_input)
113 {
114     while(1){
115         HAL_GPIO_TogglePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin);
116         HAL_Delay(500);
117         tx_thread_sleep(20);
118     }
119 }
120 /* USER CODE END 1 */
```

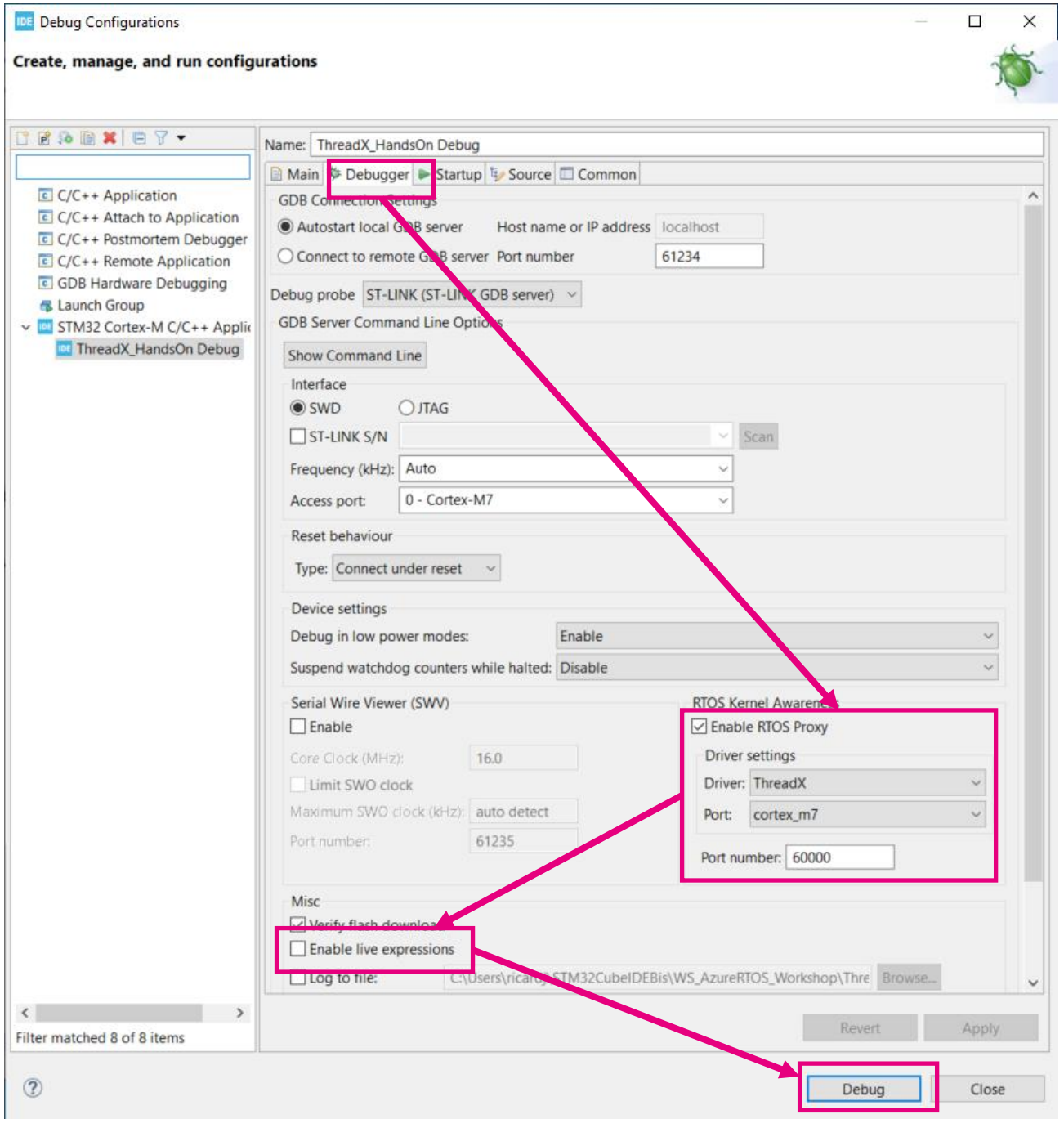
life.augmented

Expected behavior



*) HAL_Delay() is interrpted by higher prio thread which just become active

**) Continue with the interrpted HAL_Delay() which timer now has elapsed

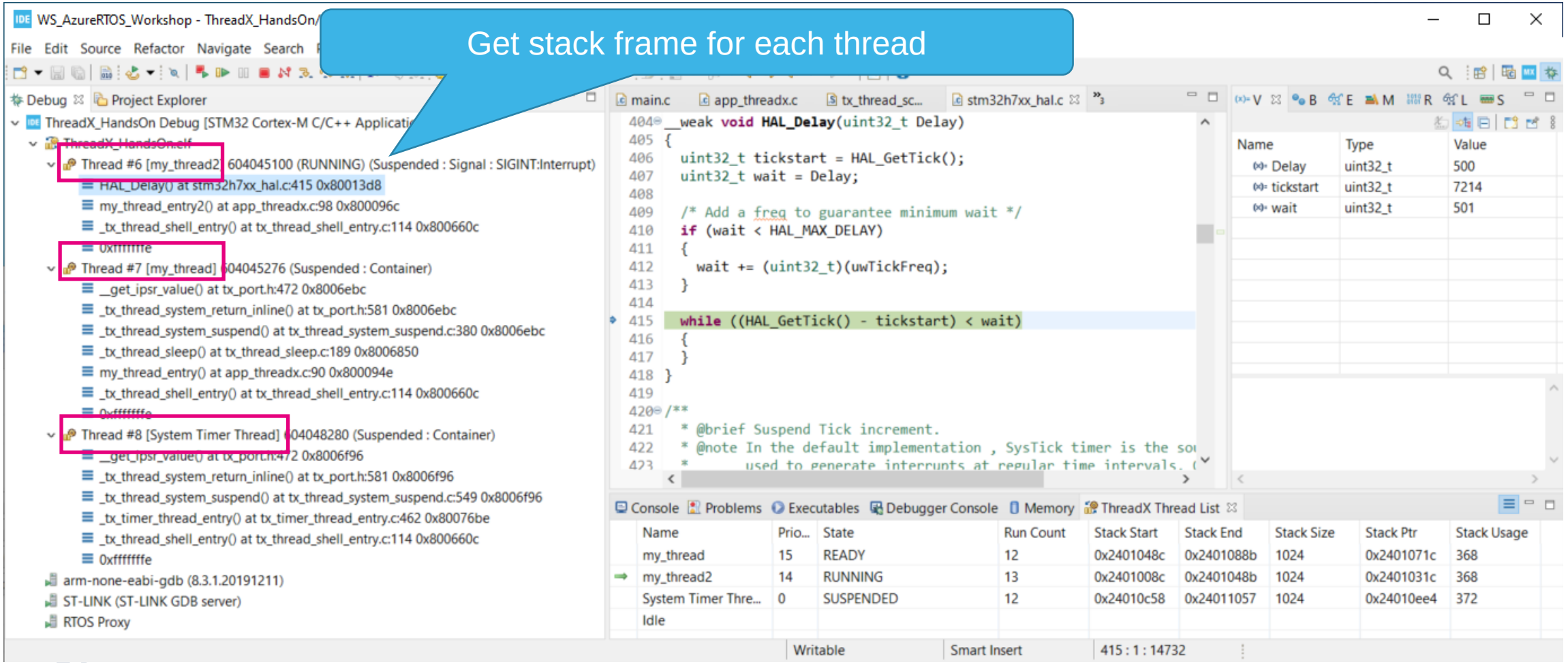


Debugger ThreadX aware

- Open Run/Debug Configuration
- Select Debugger tab
- Enable RTOS Proxy
- Select Cortex M7
- Unselect ‘Enable Live Expression’
- Click on Debug

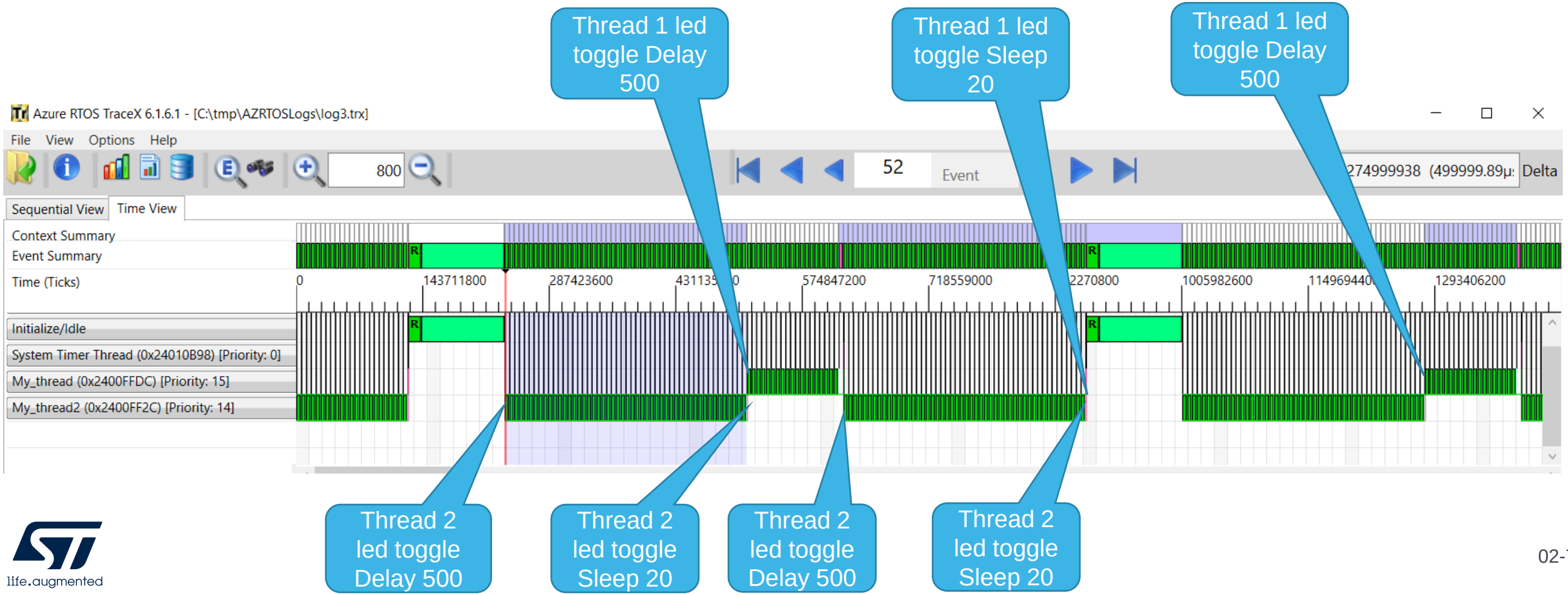
ThreadX Stack frames

- Execute the firmware for few seconds. 2 LEDS should blink with different periods



Get Trace

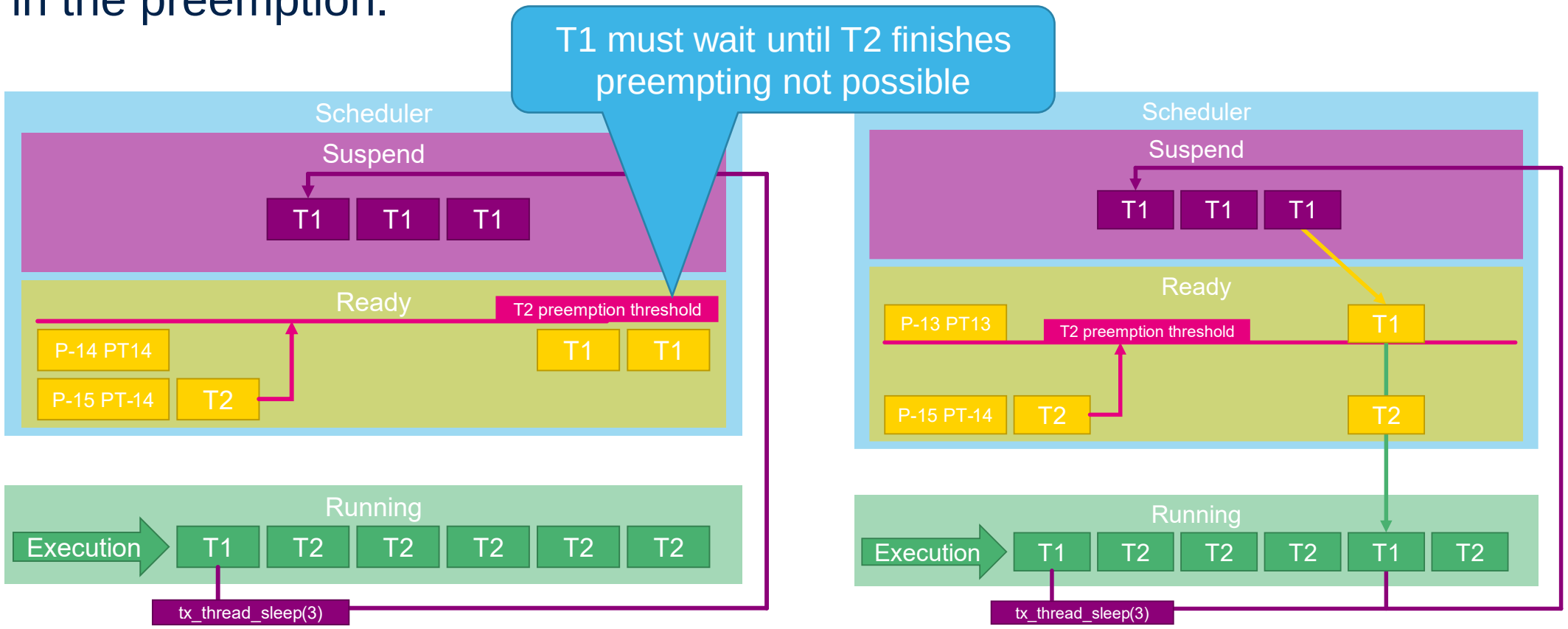
- Go to memory tab and export trace buffer to log3.trx
- Switch to TraceX, open log3.trx, time view, unzoom



life.augmented

Preemption threshold principle

- Preemtion threshold is a ThreadX mechanism that introduces a kind of hysteresis in the preemption.

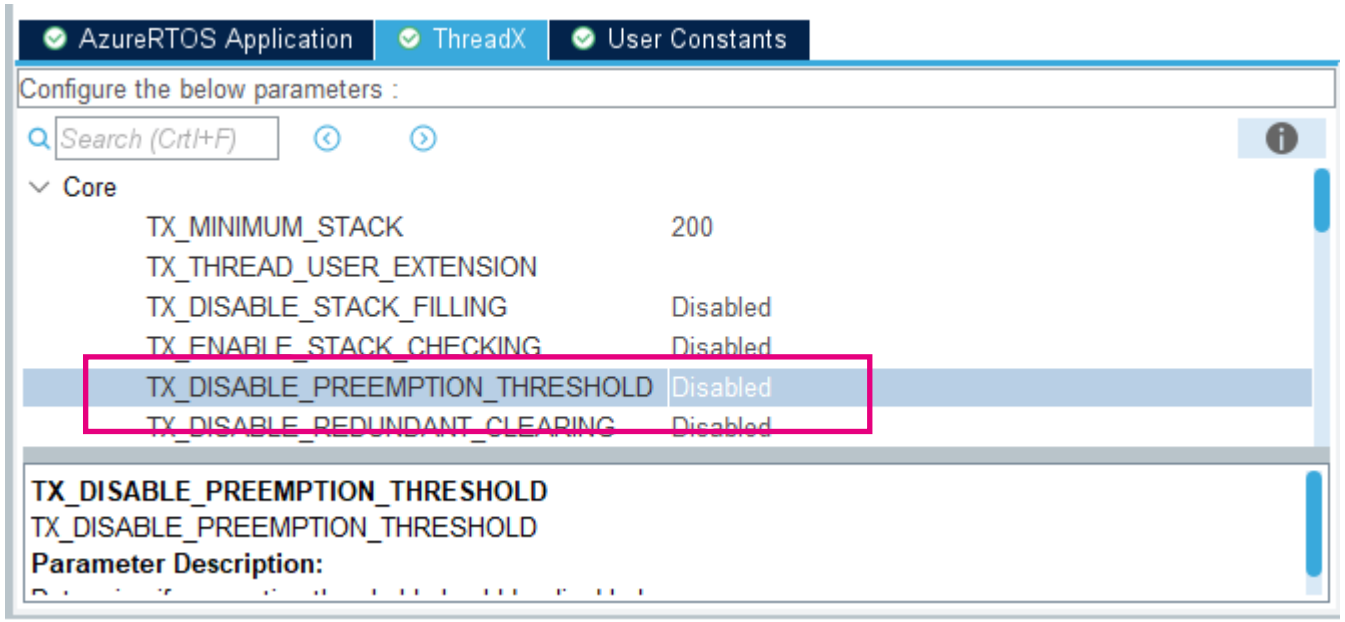


Activation of preemption threshold feature

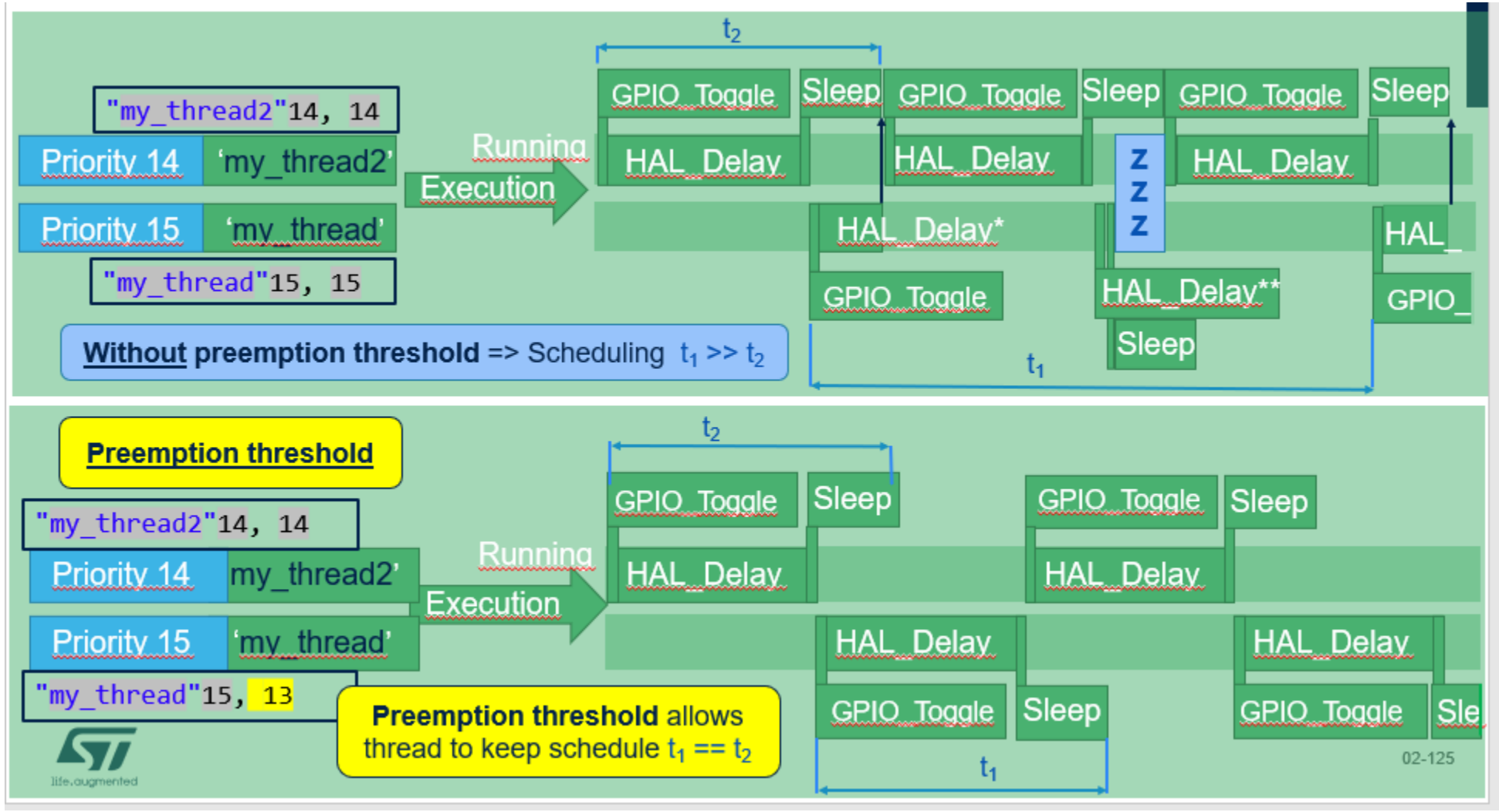
Core\Src \app_threadx.c

- Enable this feature in CubeMX
  - Set TX_DISABLE_PREEMPTIN_TRESHOLD to DISABLE
  - ALT-K to update code
- Change my_thread preemption priority to 13

```
tx_thread_create(&thread_ptr, "my_thread",
      my_thread_entry, 0x1234,
      thread_stack, THREAD_STACK_SIZE,
      15, 13, 1,
      TX_AUTO_START);
```



Behavior of preemption threshold



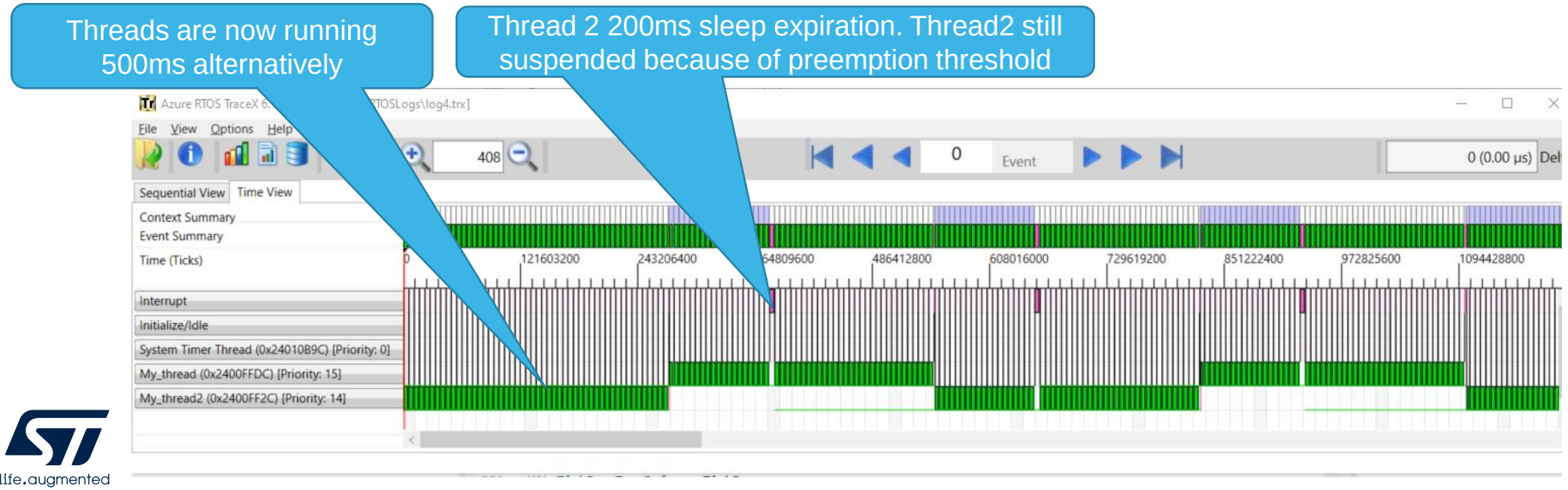
*) HAL_Delay() is interrpted by higher prio thread which just become active

**) Continue with the interrpted HAL_Delay() which timer now has elapsed

life.augmented

Check with TraceX

- Launch the debugger and execute the firmware for few seconds
- LEDs should blink alternatively with same period
- Go to memory tab and export trace buffer to log4.trx
- Switch to TraceX and open log4.trx, time view, unzoom



life.augmented

Conclusion

- We have seen how ThreadX is integrated in our environment
- It is easy to add ThreadX in a project and start using ThreadX API in an already prepared project structure
- STM32CubeIDE provides ThreadX specific views for debugging
- TraceX usage

Step 4 : CMSIS_OS2 wrapper

CMSIS_OS2 wrapper

- Can be added manually to project
- Copy cmsis_os2.h and cmsis_os2.c to your project
  - Copy cmsis_os2.c from Repository\Packs\STMicroelectronics\X-CUBE-AZRTOS-H7\2.0.0\Middlewares\ST\cmsis_rtos_threadx\
  - Copy to cmsis_os2.h from Repository\STM32Cube_FW_H7_V1.9.0\Drivers\CMSIS\RTOS2\Include\
- #Include “cmsis_os2.h” to your project
- Modify tx_user

```
#define TX_MAX_PRIORITIES                64
#define TX_THREAD_USER_EXTENSION         ULONG tx_thread_detached_joinable;
```

- You can now use CMSIS_OS2 API
- Check README.md for api supported by port

Step 2 : Short ThreadX features overview

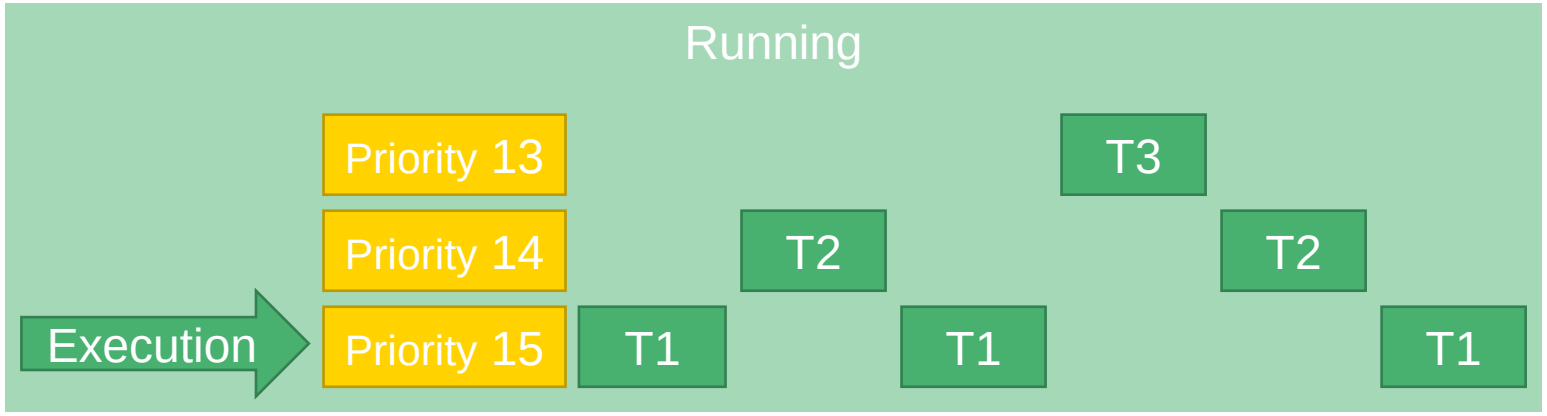
life.augmented

List of components

- Threads
- Memory pools
- Semaphores
- Mutexes
- Queues
- Events
- Timers

Thread

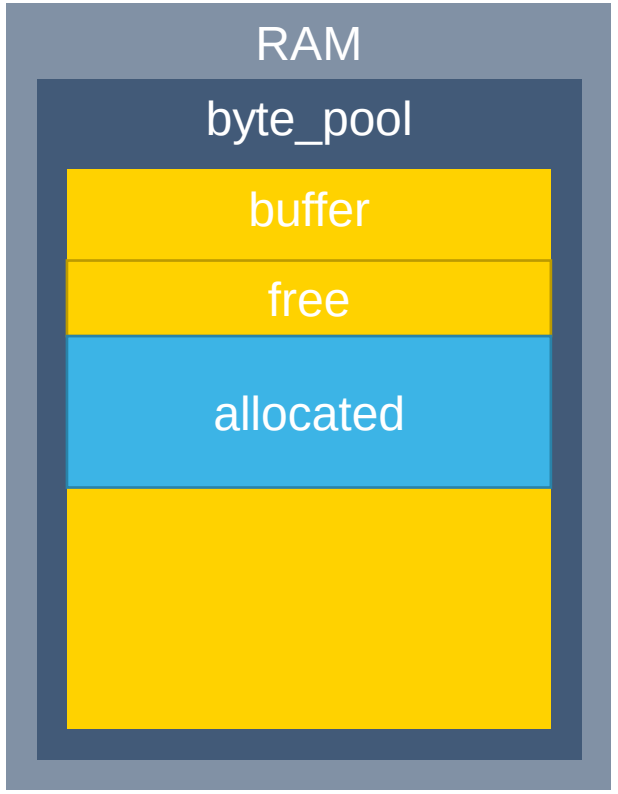
- Threads
  - Preemption priority
  - Priority threshold
  - Time slicing

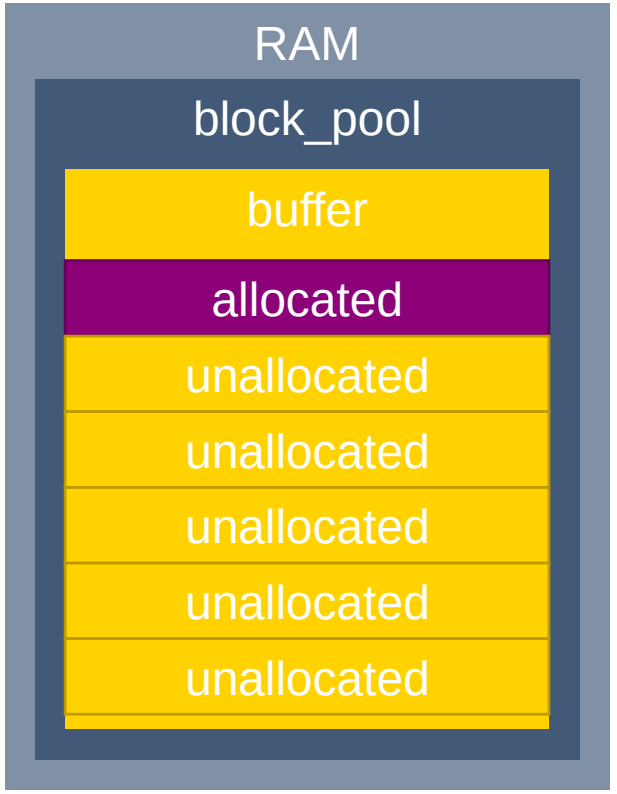


https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter3#thread-execution-1
https://rristm.github.io/stm32_threadx/show/thredx_thread.md

Memory pool

- Byte pool
  - Can allocate any value in bytes
  - Can be fragmented back
  - Fragmentation issues
- Block pool
  - Can give only one block fixed size
  - No fragmentation problems

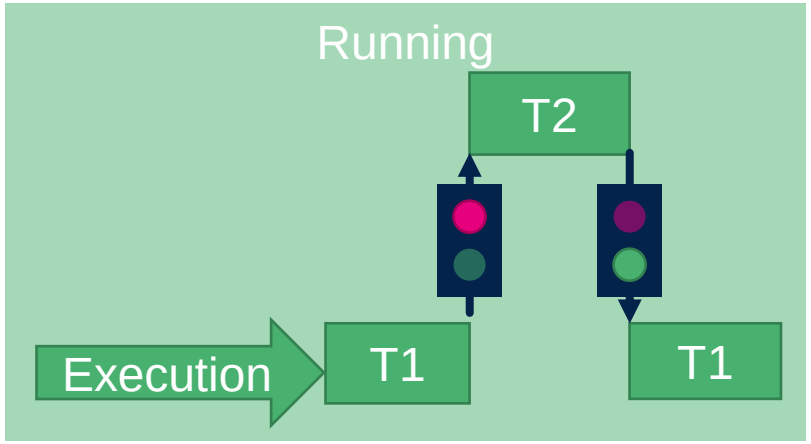




https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter3#memory-byte-pools
https://rristm.github.io/stm32_threadx/show/threadx_memory_pools.md

Semaphore

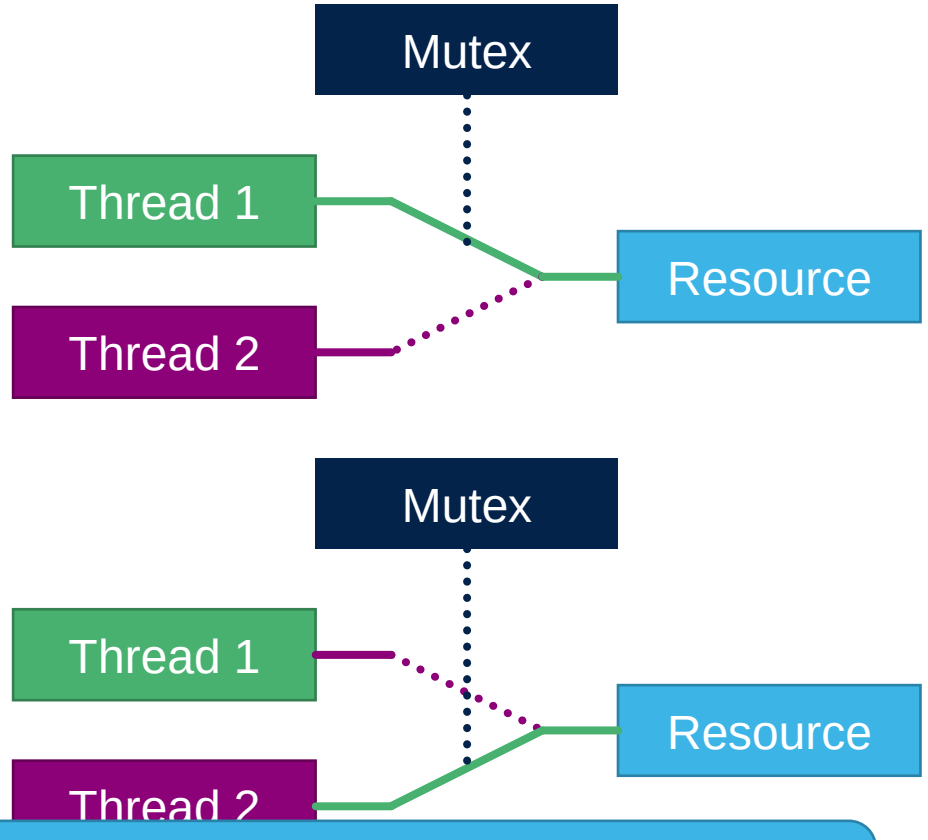
- Semaphore
  - Work as counting semaphore
  - Can work as binary semaphore



https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter3#counting-semaphores
https://rristm.github.io/stm32_threadx/show/thredx_semaphore.md

Mutex

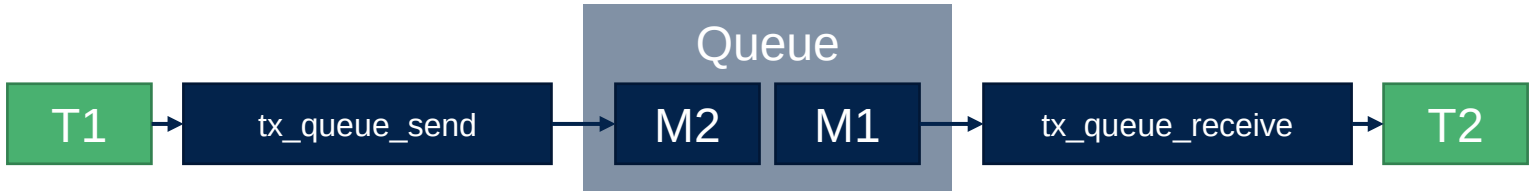
- Mutex
  - Work as binary semaphore
  - Guards access to one resource
  - Thread can inherit priority



https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter3#mutexes
https://rristm.github.io/stm32_threadx/show/thredx_mutex.md

Queue

- Queue
  - Used to send messages between threads
  - Up to 16 words (16*32bit)

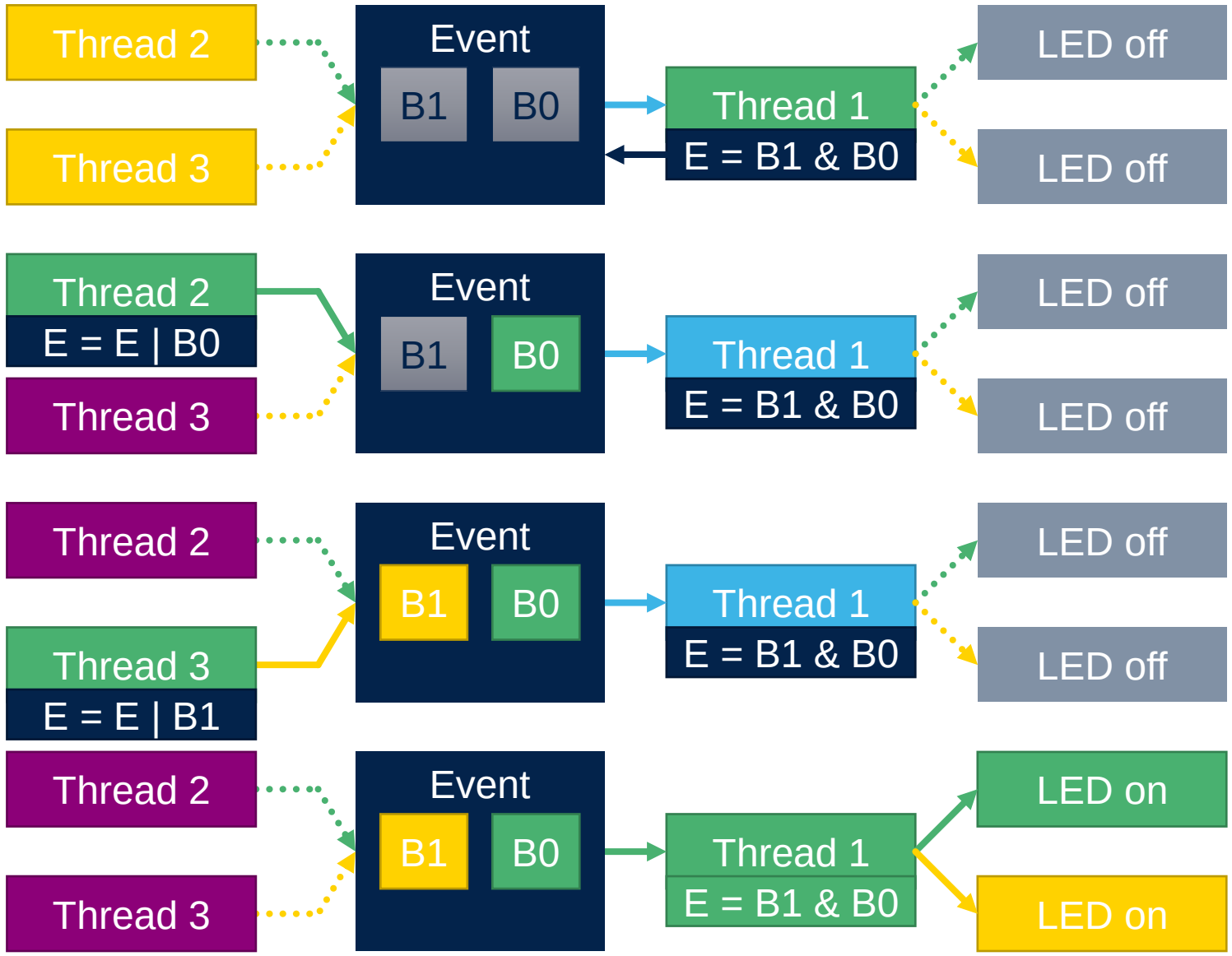


https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter3#message-queues
https://rristm.github.io/stm32_threadx/show/thredx_queue.md

Event

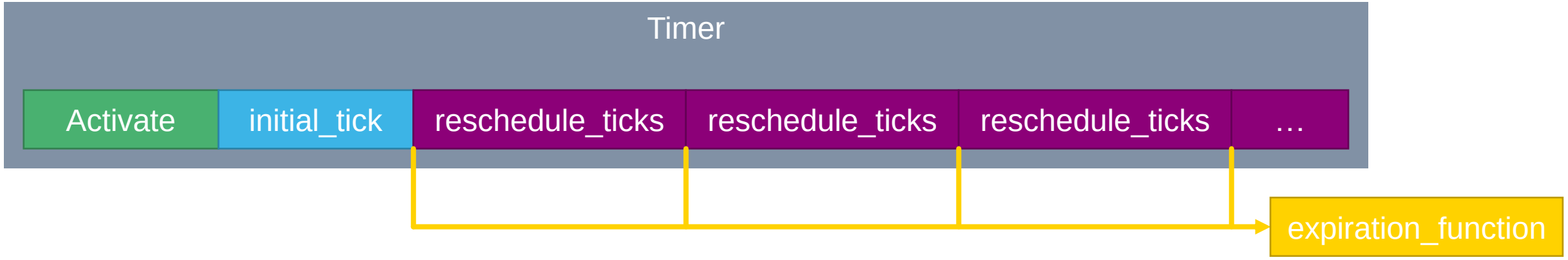
- Events
  - Used to synchronize thread
  - Base on OR / AND bit logic
  - One event have 32bits.

https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter3#event-flags
https://rristm.github.io/stm32_threadx/show/thredx_events.md



SW Timer

- SW Timers
  - SW timer based on ThreadX ticks
  - Can be used to create delays or timeouts (only RTOS ticks)
  - Repetitive and one-shot mode



https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter3#application-timers-1
https://rristm.github.io/stm32_threadx/show/threadx_timer.md

life.augmented

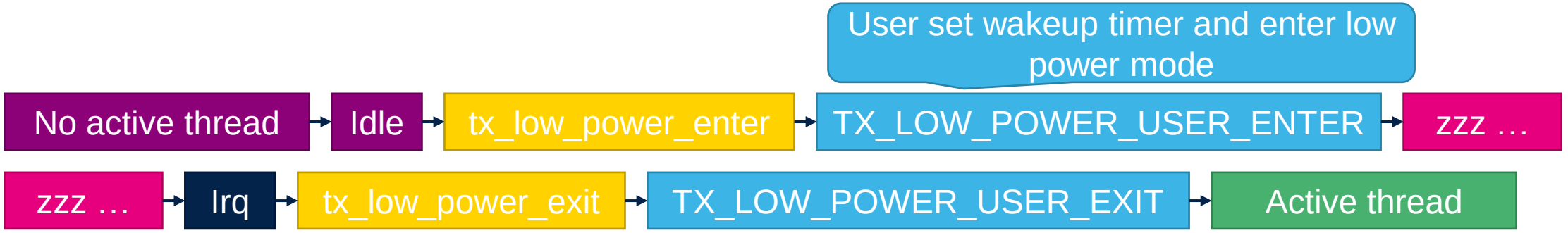
Low power

https://github.com/azure-rtos/threadx

- Introduced in ThreadX 6.1.5
  - tx_low_power functions
  - Allow to calculate time needed for low power before enter
  - Allow to set new time after low power exit
  - Resume ThreadX operation after low power

Example: X-CUBE-AZRTOS-H7\2.0.0\Projects\NUCLEO-H723ZG\Applications\ThreadX\Tx_LowPower

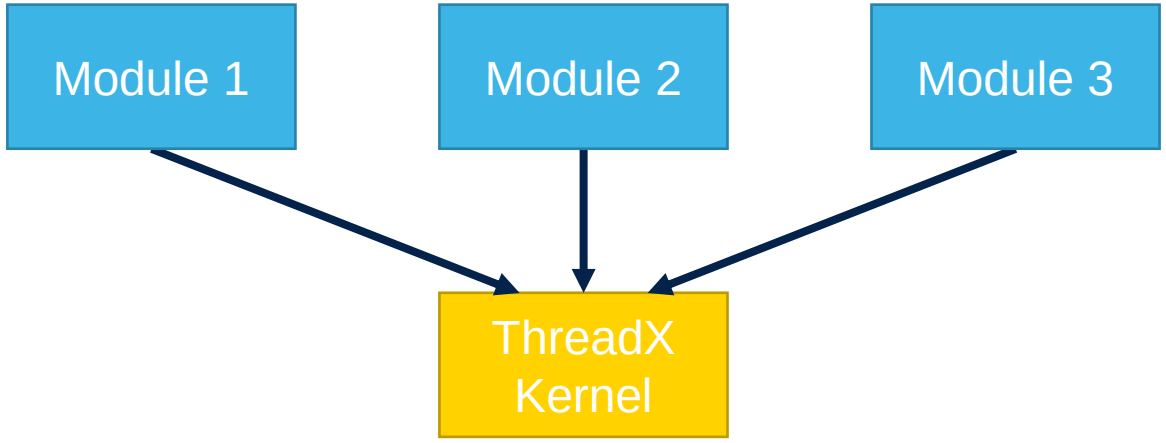
  - Activated by definition TX_LOW_POWER
  - By defining `TX_LOW_POWER_USER_ENTER` function to user defined low power enter.



life.augmented

Modules

- Application can be separated to modules
- Modules possible to load on the fly
- Modules can be protected by MPU



https://docs.microsoft.com/en-us/azure/rtos/threadx-modules/chapter1

ThreadX footprint

- https://docs.microsoft.com/en-us/azure/rtos/threadx/overview-threadx

Azure RTOS ThreadX Service	Typical Size in Bytes
Core Services (Require)	2,000
Queue Services	900
Event Flag Services	900
Semaphore Services	450
Mutex Services	1,200
Block Memory Services	550
Byte Memory Services	900

Performance info

Performance info

- Allow to get performance statistics inside app
- Functions have *_performance_* in their name
- Here link how to enable it

As and example. *performance_system_info will give information's about system states

how many times threads were resumed for example

```
tx_thread_performance_system_info_get(&resumptions,
            &suspensions,
            &solicited_preemptions, &interrupt_preemptions,
            &priority_inversions, &time_slices, &relinquishes,
            &timeouts, &wait_aborts, &non_idle_returns,
            &idle_returns);
```

Adding this feature will increase ThreadX size

Thank you

life.augmented