21-22: RTOS Introduction, Threads and Delays

v3

Big Picture on Task Scheduling and Operating System (part 1)

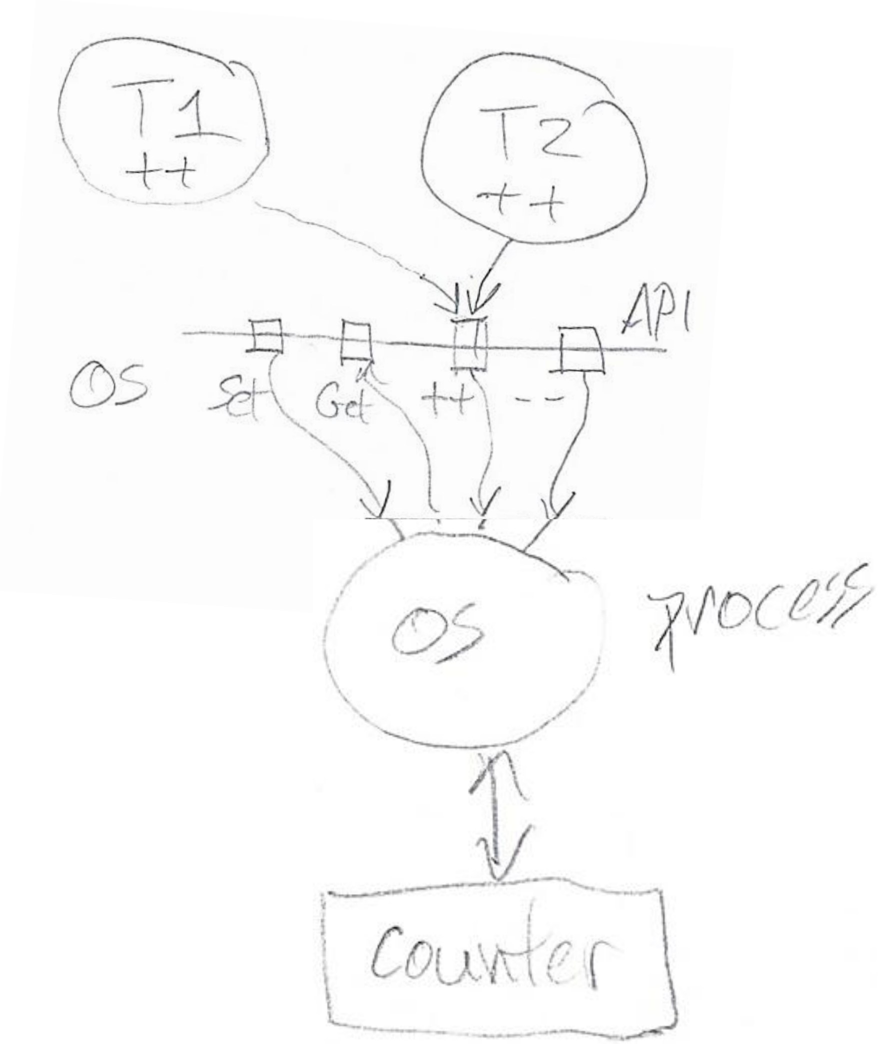
- Selecting and running tasks is main duty of operating system
  - Again, task == thread here. Type of software process

- What else can OS do for us?
  - Some features in scheduler make it easy to add **new capabilities**

- Task Preemption…
  - Supporting task preemption and resumption requires saving and switching task contexts
  - Enables **Cooperative Scheduling:** Task can **voluntarily yield CPU** to scheduler (“Run something else now. Later on, resume running me here.”).
  - Run-to-completion behavior not required for tasks. Easier, since don’t need to turn task code into finite state machine to share CPU with other tasks **sooner** (for **responsiveness**) or for **longer** (for less time **overhead**)
  - ISRs must still be RTC.

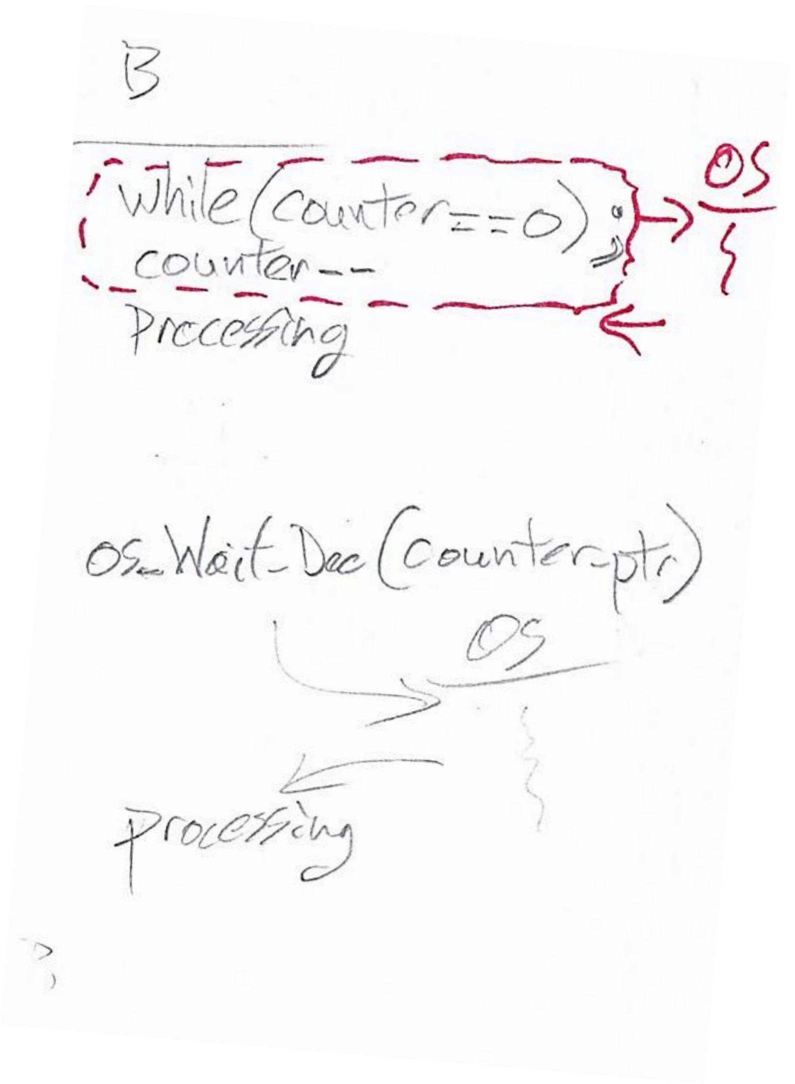
Part 2: Provide Protected Shared Variables (and more)

- OS can serve as **centralized manager** of variables, other resources for tasks to share
- Provide tasks with **special shared resources** (e.g. variables): **protected** from race conditions.
  - Tasks access these shared resources **only** through specific OS calls. Example for integer counter variable:
    - Set value
    - Get value
    - Increment
    - Decrement
  - OS protects critical sections by **serializing access** to a shared variable and **preventing unsafe preemption**.
    - Don't let task A access shared variable if task B has started accessing it but not finished yet.
    - Protection of critical section code implemented in OS so you don't have to do it

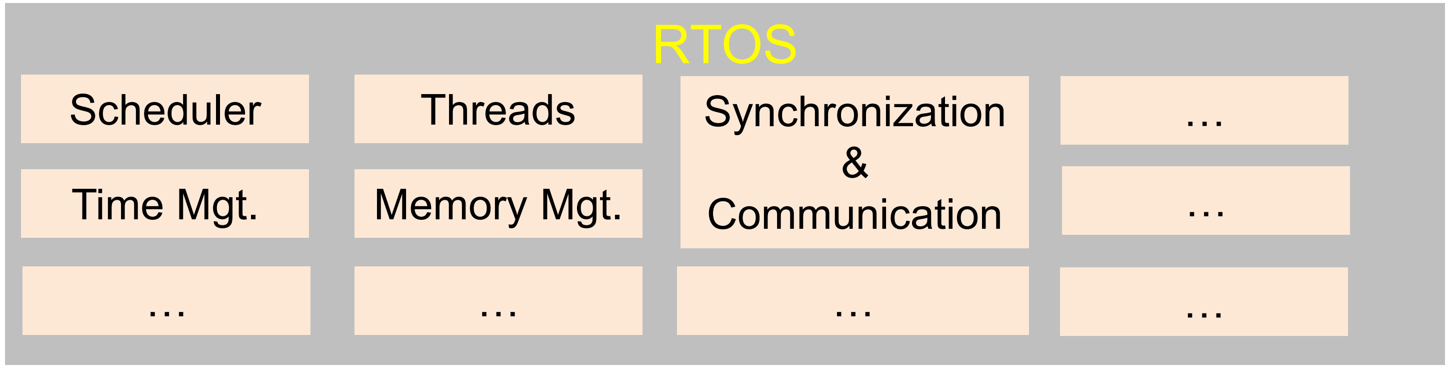


Part 3:
Enhance Shared Counter Variable to Synchronize Task Execution

- Add a few rules on **how process uses the counter**. Semaphore example:
  - Task A detects event, then signals event has happened by **incrementing counter**
  - Task B loops: **wait until counter > 0**, then **decrement counter** and **do processing** in response to event.
- Add an OS Call for counter variable used as semaphore
  - Task B calls OS
    - `Wait until counter > 0, then decrement counter and let me continue`
  - OS processing affects both variable and task state (and therefore task scheduling)
    - `If counter == 0, then make task wait: change task state from ready to blocked. Do other tasks until counter > 0.`
    - `Next, decrement counter and let task resume: change state from blocked to ready.`
- Now OS **schedules tasks better** while **simplifying development**
- Tweak rules for **mutually exclusive execution** of task code critical sections
- Extend concept, apply elsewhere: OS provides **inter-process synchronization and communication** (IPC) primitives and useful data structures (e.g. message queues) to application processes



RTOS: What and Why



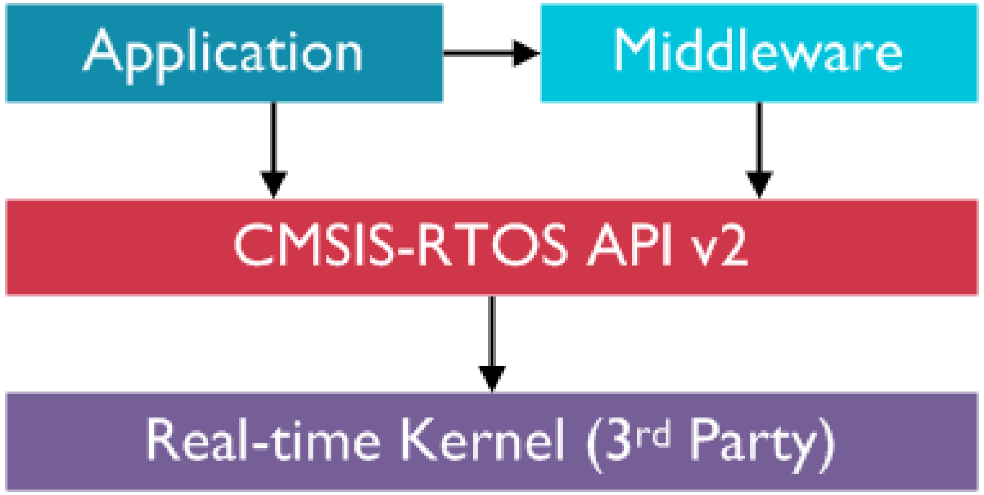
- Real-Time Operating System
  - An OS designed to operate with **deterministic** (repeatable) timing
    - Typically uses **preemptive task scheduling** for better responsiveness
    - Timing: Deterministic, predictable, bounded
- Why use one? RTOS vs. OS
  - Easier to build a system with deterministic timing
    - Developer can more easily manage the response times of urgent processing through prioritization
    - Don't need to restructure code repeatedly or re-invent the wheel (*hopefully correctly*)
  - Cutting response time reduces processor & memory speed requirements (and HW $$)
- Why else? RTOS or OS vs. bare-metal
  - Improve software modularity
    - Improve software reliability by isolating threads
    - Simplify maintenance and upgrades
  - Leverage built-in OS/RTOS services
    - Interprocess communication and synchronization (safe data sharing)
    - Time management
    - I/O abstractions
    - Memory management
    - File system
    - GUI
    - Networking support

Keil RTX5 Introduction

- Open source real-time kernel from Keil (part of ARM)
- Names
  - RTX = Real-Time Executive
  - Starting with CMSIS version 6, RTX5 is also called CMSIS-RTX
- Documentation online
  - https://www.keil.com/pack/doc/CMSIS/RTOS2/html/rtx5_impl.html
- Included with MDK, supports integrated kernel debugging
- Configurable to minimize use of memory
- Supports multiple types of scheduling
  - Preemptive
  - Cooperative (non-preemptive)
  - Time slicing
- Provides various services
  - Thread Management
  - Time Management
  - System Control
  - Thread Synchronization and Communication
  - Memory Management

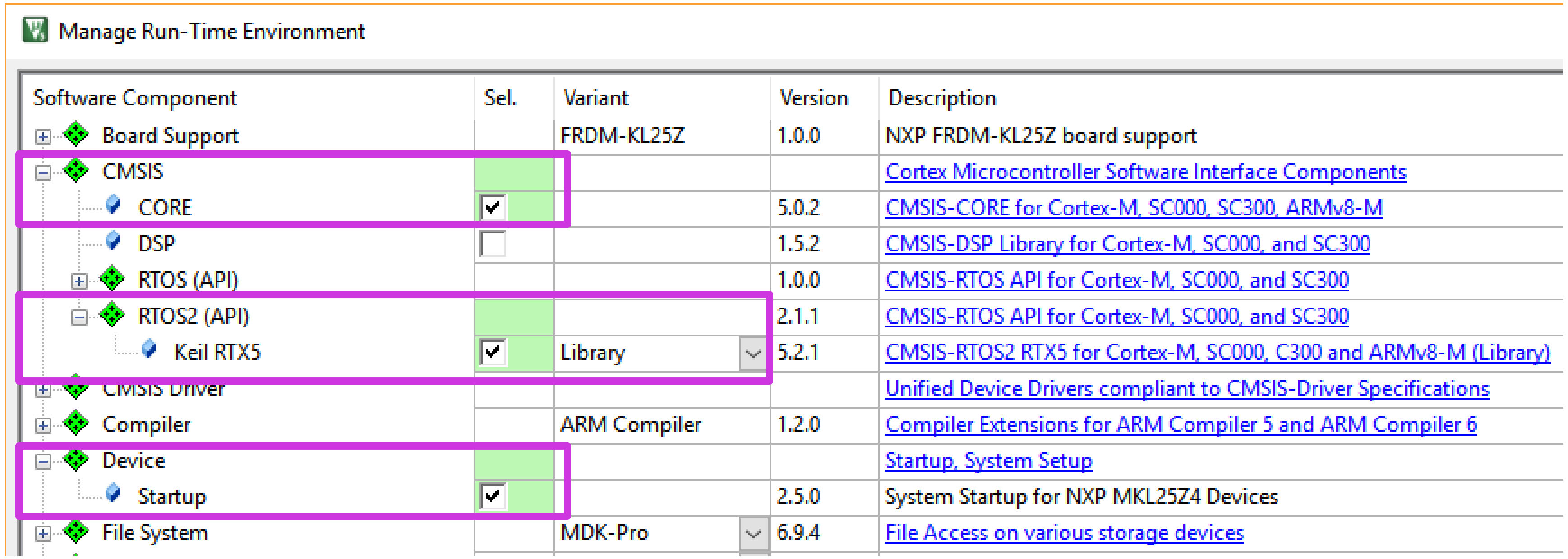
CMSIS-RTOS2

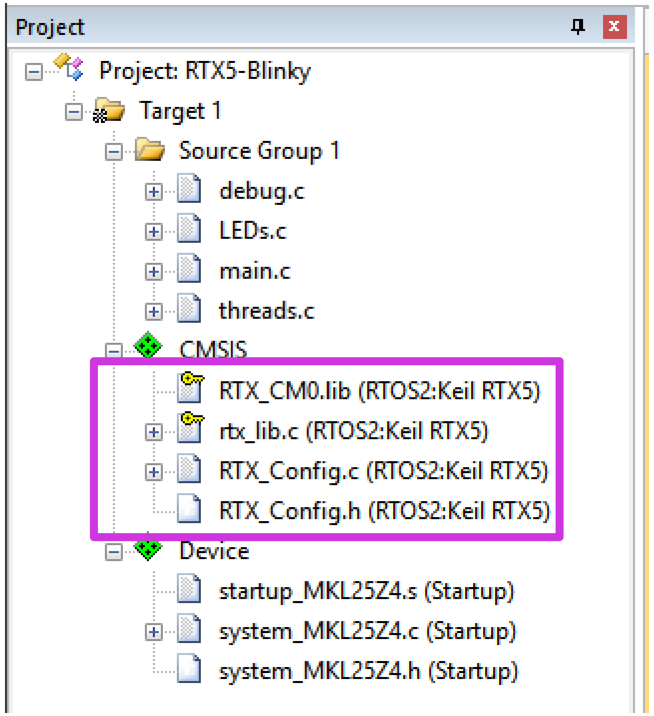
- CMSIS = Cortex Microcontroller Software Interface Standard
  - Conventions and standards for software interfaces, structure and names
  - Hardware Abstraction Layer: Software layer between application program and hardware
- CMSIS-RTOS2:
  - API which provides standardized interface to different RTOSs
    - RTX5, FreeRTOS, µC/OS, etc.
  - Documentation:
  - https://arm-software.github.io/CMSIS_6/latest/RTOS2/index.html
- Your source code must #include “cmsis_os2.h”



CMSIS-RTOS2 API Structure
(from CMSIS documentation, ARM Ltd.)

Using RTX5 in MDK



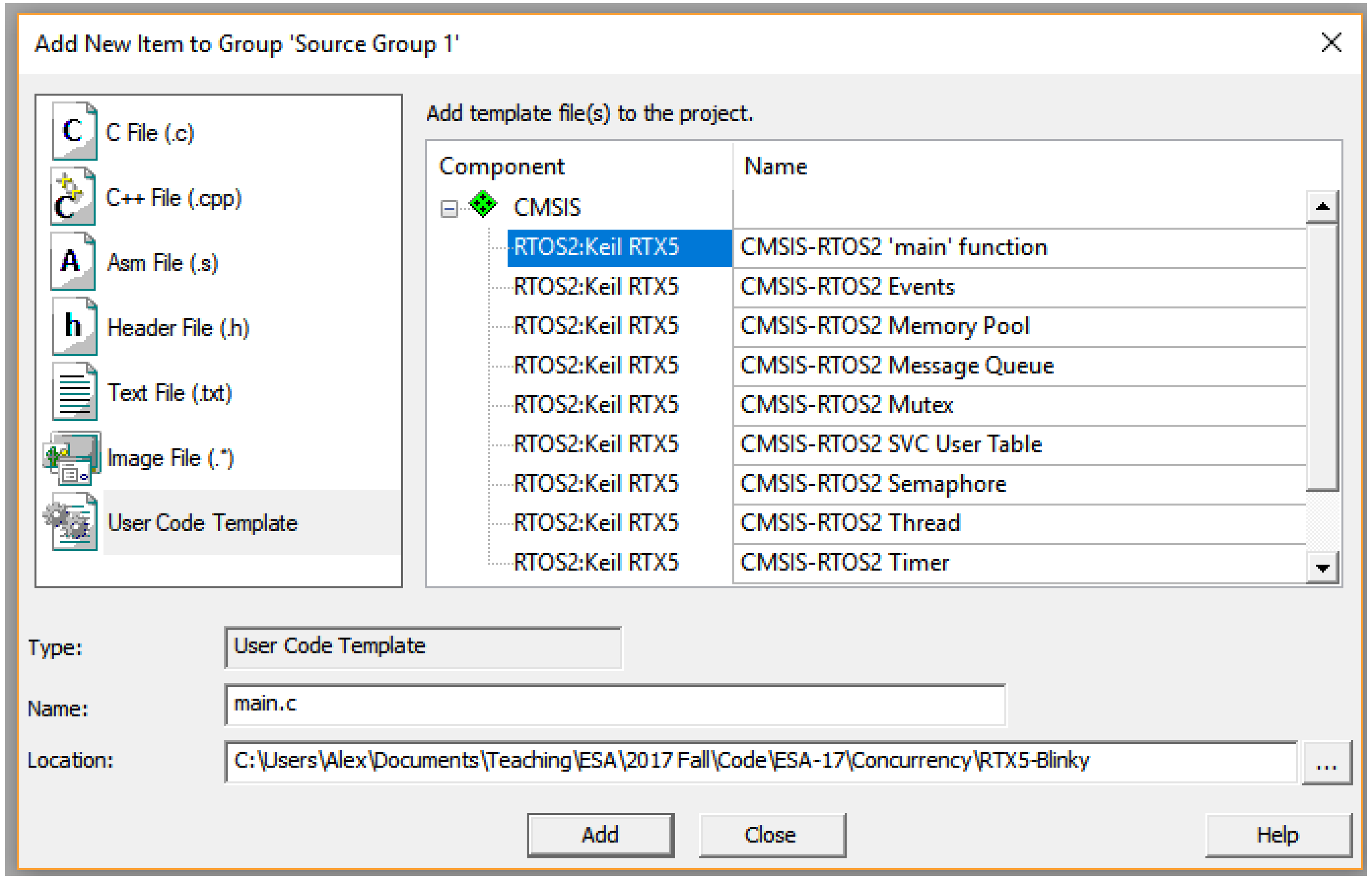


- When creating a new project, select Run-Time Environment Software Components
  - CMSIS ⟶ Core
  - CMSIS ⟶ RTOS2 (API) ⟶ Keil RTX5 (Library, not Source)
  - Device ⟶ Startup
- Precompiled RTX code is located in RTX_CM0.lib

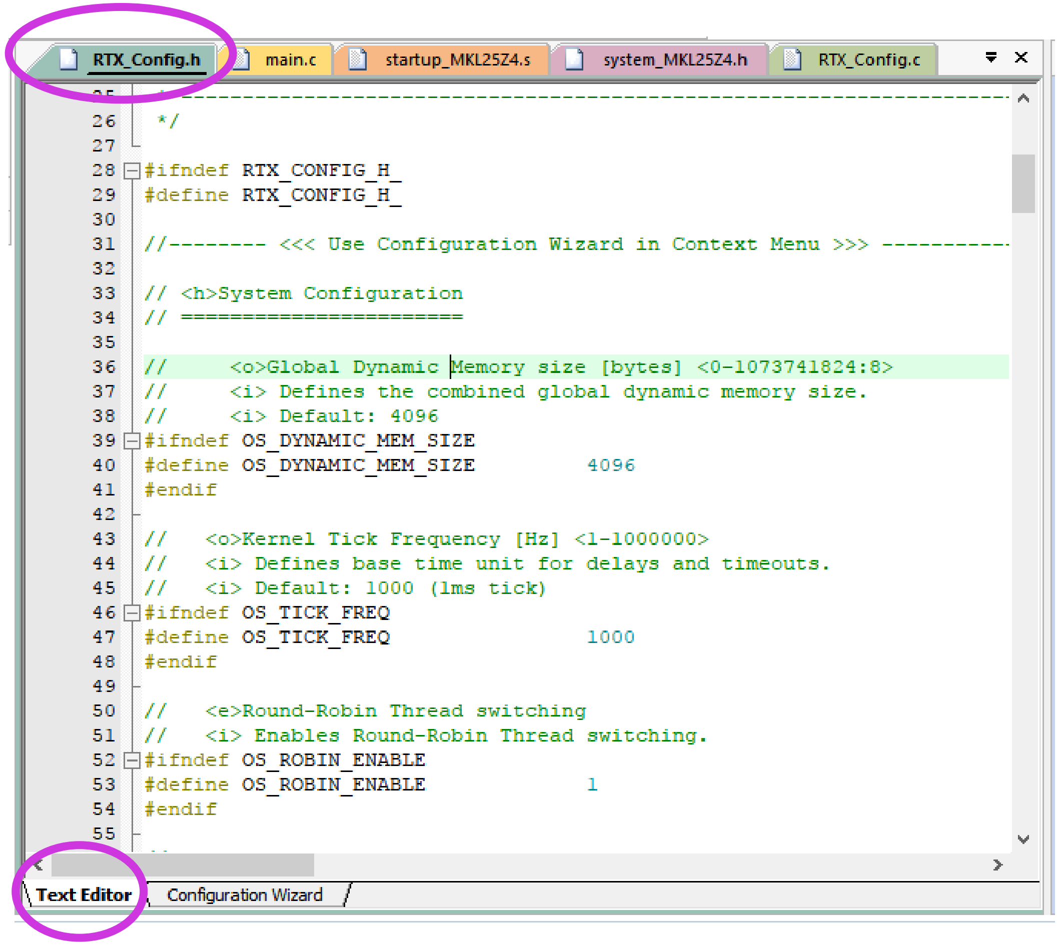
- RTX_Config.c and .h are copied to your project (in RTE\CMSIS) for your modification
- Need to include header file in all source files which use the RTOS
  - #include "cmsis_os2.h"

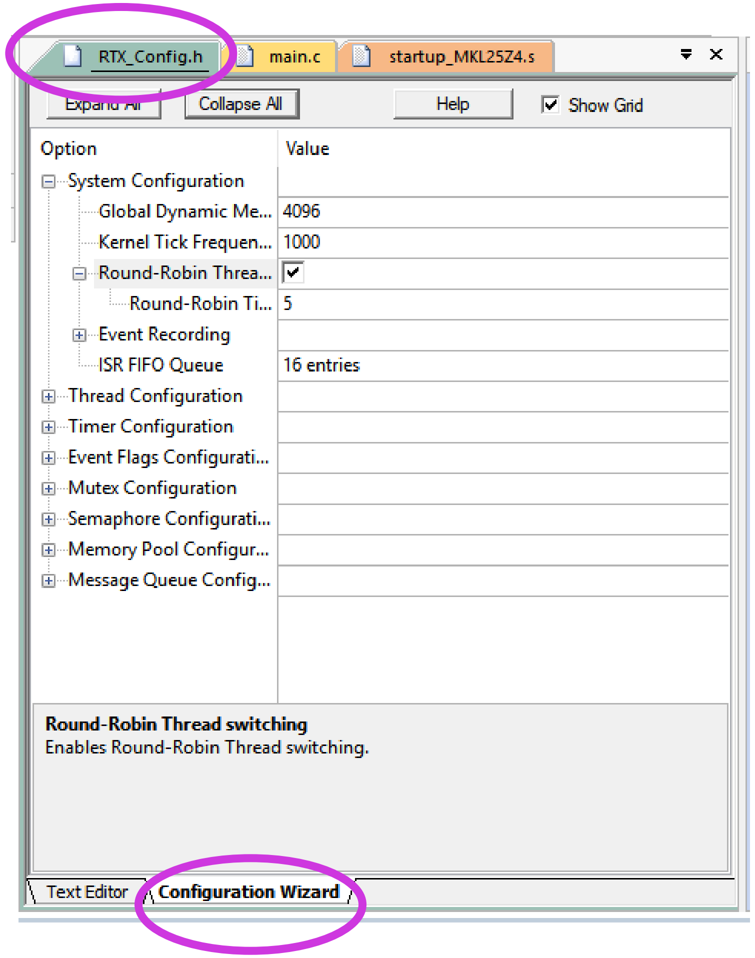
Using Source Code Templates

- Right-click on Source Group 1 -> Add New Item to Group…
- Select User Code Template, expand CMSIS node
- Select CMSIS-RTOS2 'main' function
- Optional: change target location (at bottom) to a Source folder if present
- Click Add



RTX5 Configuration

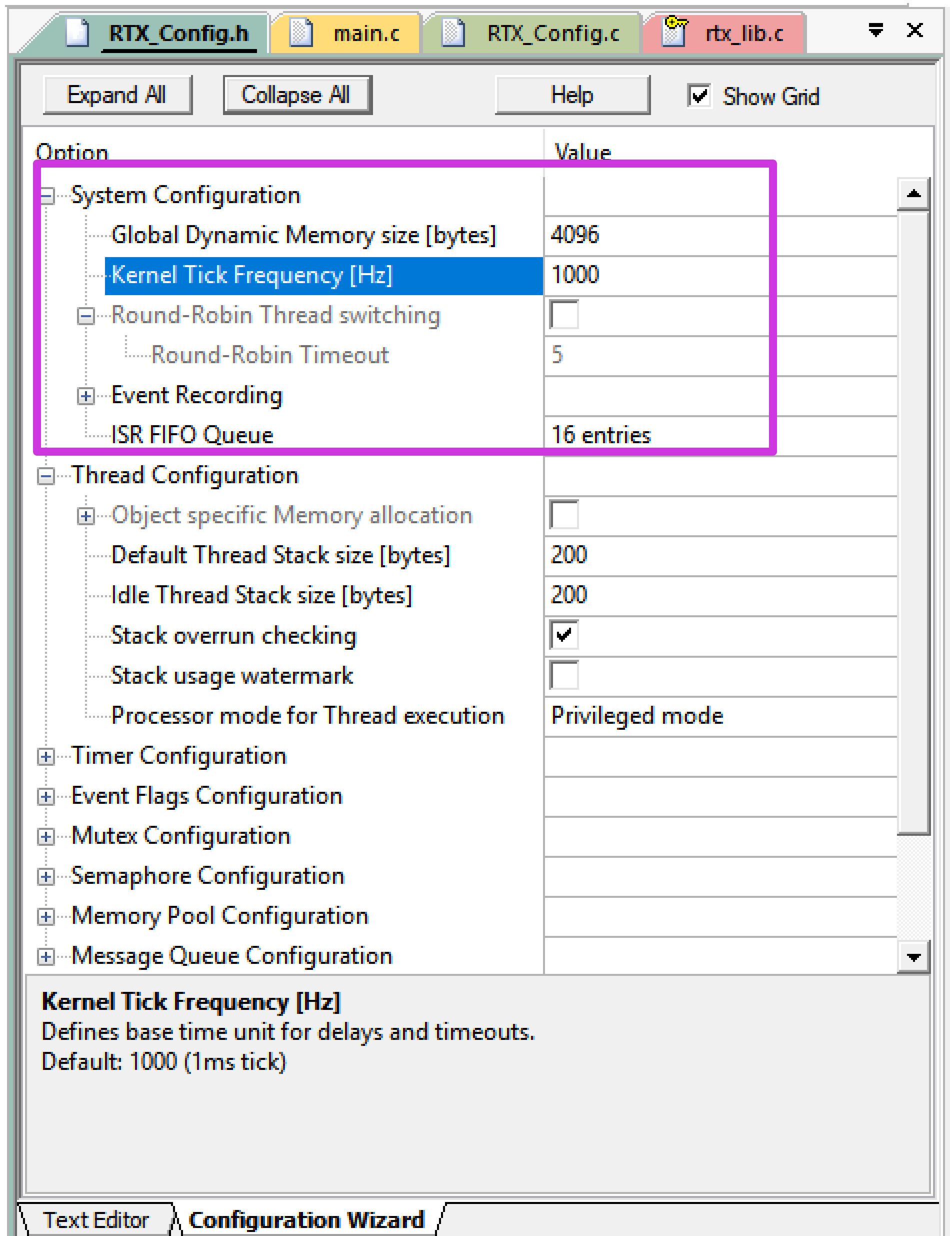




- Modify RTX_Config.h

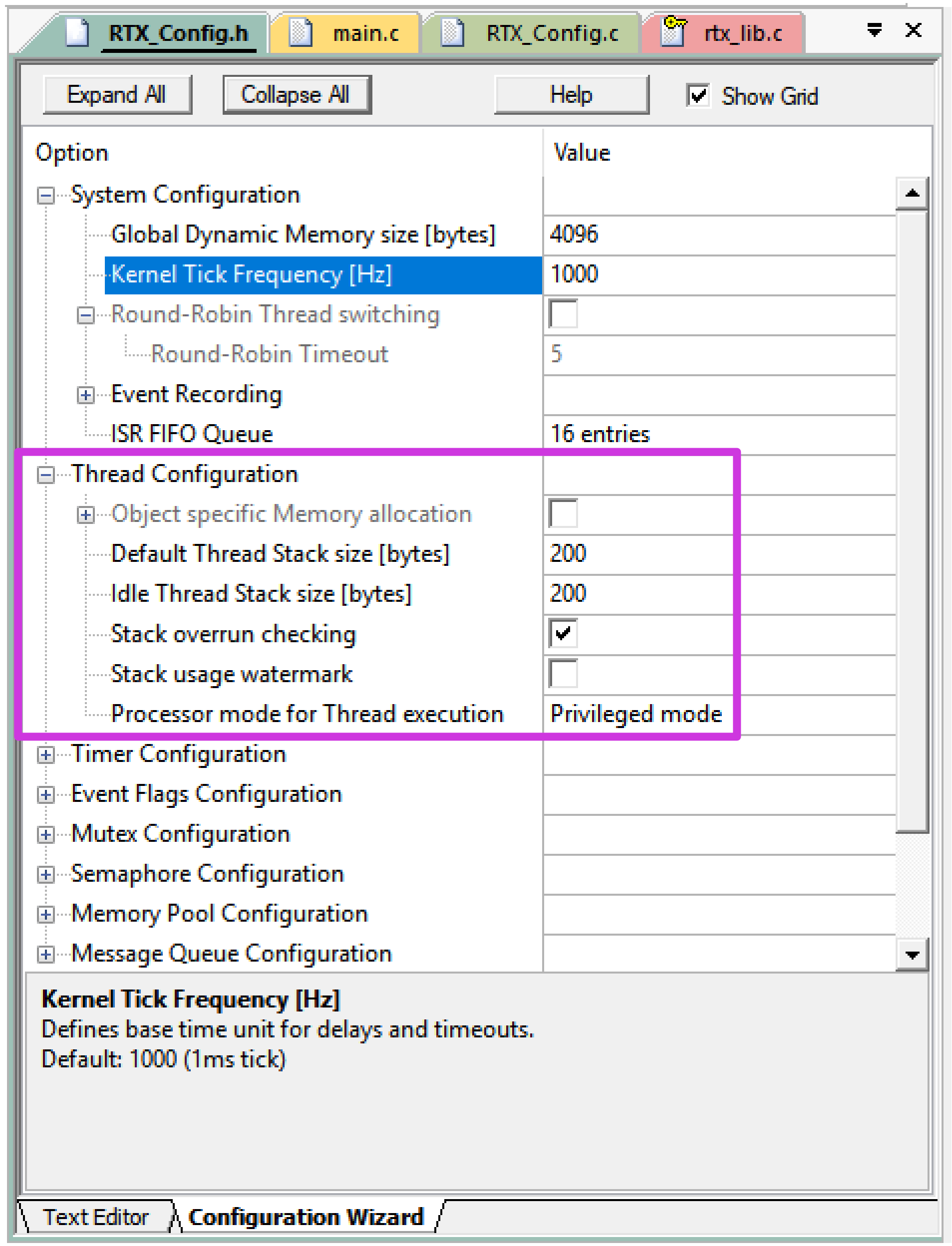
System Configuration and Clock Tick

- OS needs a periodic interrupt
  - Supporting time delays and timeouts
  - Supporting time slicing (if used)
  - Uses SysTick timer on MCU
- Configurable in RTX_Config.h
  - Kernel tick frequency determines timer clock settings
  - What frequency? Trade off overhead vs. response time and accuracy of delays
- Each tick invokes thread manager
  - Highest-priority thread, performs scheduling and other system work
- Disable (uncheck) Round-Robin thread switching for true prioritized scheduling



Thread-Related Configuration

- Thread stack sizes
- Support for monitoring stack size
  - Overflow (overrun) detection
  - Watermark initialization
- Processor mode when executing threads



Debug Support

- View->Watch->RTX RTOS
- View->Periodic Window Update

- *Debug->OS Support->System and Thread Viewer doesn't seem to work with RTX5*

- Instead, try Event Recorder

RTX RTOS

Property	Value
System	
Kernel ID	RTX V5.2.1
Kernel State	osKernelRunning
Kernel Tick Count	15763
Kernel Tick Frequency	1000
Round Robin Tick Count	0
Round Robin Timeout	5
Global Dynamic Memory	Base: 0x1FFFF000, Size: 4096
Stack Overrun Check	Enabled
Stack Usage Watermark	Disabled
Default Thread Stack Size	200
ISR FIFO Queue	Size: 16, Used: 0
Threads	
id: 0x200002B4, osRtxIdleThread	osThreadReady, osPriorityIdle, Stack Used: 32%
State	osThreadReady
Priority	osPriorityIdle
Attributes	osThreadDetached
Stack	Used: 32% [64]
Flags	0x00000000
id: 0x200002F8, osRtxTimerThread	osThreadBlocked, osPriorityHigh, Stack Used: 52%
State	osThreadBlocked
Priority	osPriorityHigh
Attributes	osThreadDetached
Waiting	Message Get, Timeout: osWaitForever
Stack	Used: 52% [104]
Flags	0x00000000

BASIC THREAD CONCEPTS, CREATION AND DELAYS

Basic Scheduler Operation

- Scheduler runs **highest-priority ready** thread
- What if no threads are ready?
  - RTX Scheduler runs **osRtxIdleThread**, which contains infinite loop
    - Source code in RTX_Config.c
    - Can put MCU into sleep mode here to save power
      - Low Power RTX extension provides os_suspend and os_resume functions to sleep as long as possible
  - Other schedulers may keep looping, looking for ready tasks (e.g. RTCS)

- How does the scheduler know when a thread should start or stop *waiting*?
  - A thread or ISR calls an RTOS function which indicates this: signaling an event, pending on a message, etc.
  - Scheduler spins in a loop waiting for something to happen
- If one thread is running and a higher-priority thread unblocks, when is the thread switch?
  - Preemptive RTOS: immediately
  - Non-preemptive RTOS: when the lower-priority thread blocks

Thread Structure and Creation

- "Root" function is entry point to thread
  - void pointer argument, void return type
  - Is "main" function for that type* of thread
    - * Can have multiple threads with same root function, each proceeds independently and has its own thread-local variables

```
void My_thread(void const * argument) {
  // do initialization
  while (1) {
    // wait for something
    // do the processing work
  }
}
```

- Structure of root function
  - Usually function body contains infinite loop, executed one iteration at a time
    - Each time the loop is executed, should* yield the processor to let lower priority threads execute
      - ***Wait*** for time delay or next event
      - * Depends on requirements and system design
  - Can also create/destory threads dynamically
    - Create thread before each run
    - Destroy thread by letting root function complete, or by making OS call to delete or terminate it
    - Takes more time, so less responsive, more compute cycles used.
      - **Not used in this class**

CMSIS-RTOS2 Basic Thread Management

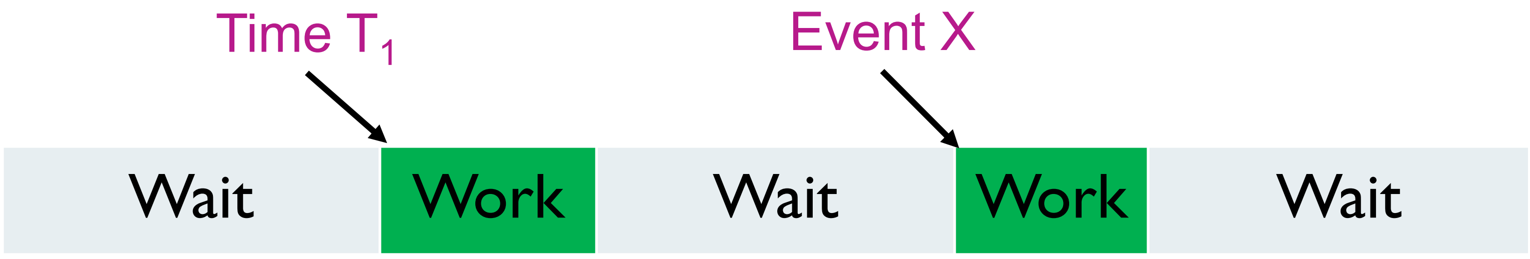
https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__ThreadMgmt.html

- Define a thread identifier variable
- Define thread root function
- Start-Up Code in main
  - Call osKernelInitialize() from main()
  - Call osThreadNew() to create additional threads. Each call returns the thread ID number.
    - Thread root function (entry point)
    - Argument
    - Attributes
  - Call osKernelStart() to start multitasking

```
osThreadId_t tid_Blinky; // Thread ID
…
void Thread_Blinky(void * argument) {
        for (;;) {
                Control_RGB_LEDs(1,1,0);
                osDelay(500);
                Control_RGB_LEDs(0,1,1);
                osDelay(500);
                Control_RGB_LEDs(1,0,1);
                osDelay(500);
        }
}
int main (void) {
  osKernelInitialize();
  tid_Blinky = osThreadNew(Thread_Blinky, arg, attrib);
  if (!tid_Blinky)
    Error_handling_code();
  else
    osKernelStart();
}
```

Triggering Threads

Thread has both working and waiting

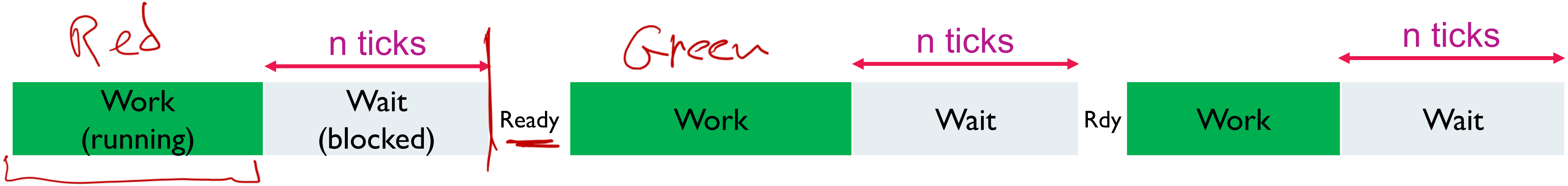


- What should the thread wait for?

- Event-triggered: thread runs when event Y happens
  - Use an OS-provided synchronization primitive (event, semaphore, queue) to signal the event has occurred
  - More details provided later

- Time-triggered: thread triggered based on time ticks
  - Very useful, saw how to do this with RTCS example
  - Can make thread **wait/block** until condition is satisfied, then make thread **ready** again
    - Wait for **num_ticks** to happen: osDelay(num_ticks)
    - Wait until tick number **abs_tick** happens: osDelayUntil(abs_tick)
  - Use hardware timer (SysTick) to generate periodic interrupt requests (events…)
  - OS uses SysTick handler to update time-related information in scheduler

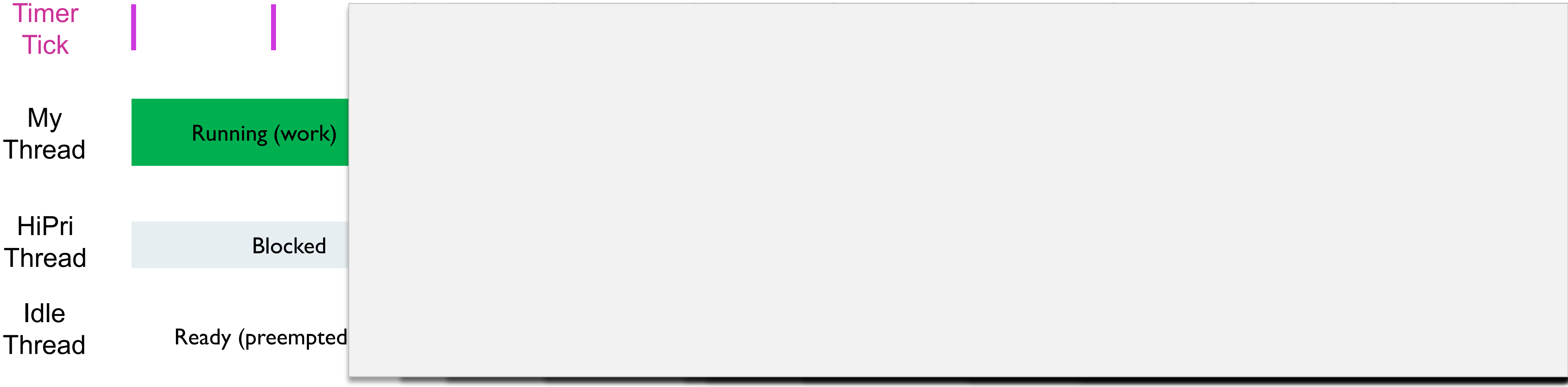
Example Program




```
void Thread_RGB(…) {
  while (1) {
      Control_RGB_LEDs(1,0,0);  <- Work
      osDelay(n);                — Wait
      Control_RGB_LEDs(0,1,0);  Work
      osDelay(n);              Wait
      Control_RGB_LEDs(0,0,1);  Work
      osDelay(n);              Wait
  }
}
```

```
void Thread_Read_Switches(…) {
  while (1) {
      if switch 1 pressed…
          do something
      if switch 2 pressed…
          do something
      osDelay(n);
  }
}
```

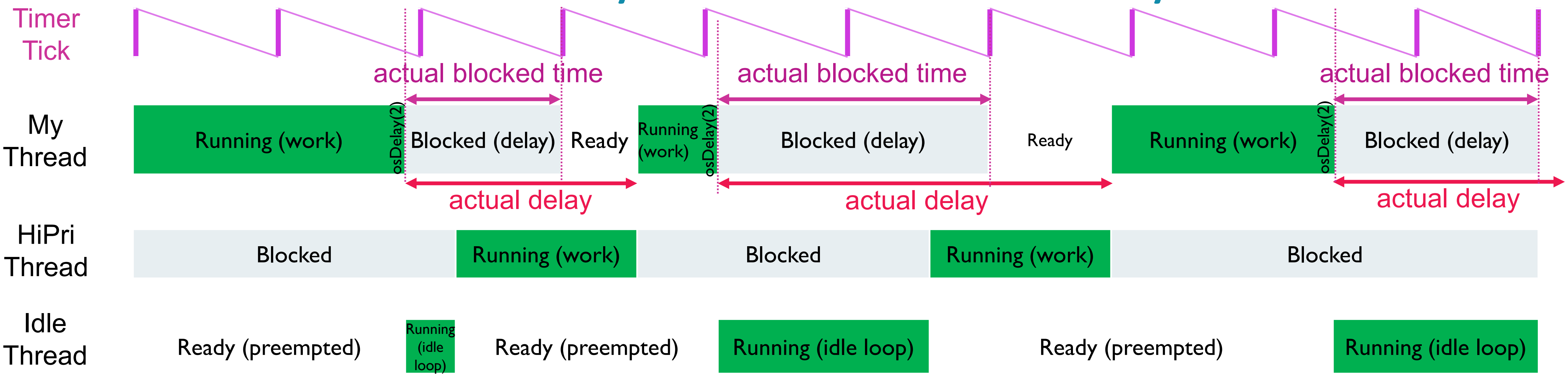
Delaying a Thread



- Delay for at least **n OS ticks** by calling osDelay(n)
  - Blocks thread until n OS ticks have occurred
  - Then thread is marked as ready to run
- Thread might not start running immediately after becoming ready
  - Higher-priority proceses (threads, ISRs) may run or already be running

```
void MyThread (…) {
  while (1) {
      // Work
      Do_Some_Work();
      // Wait
      osDelay(2);
  }
}
```

Actual Time from Delay Call to Thread Ready?

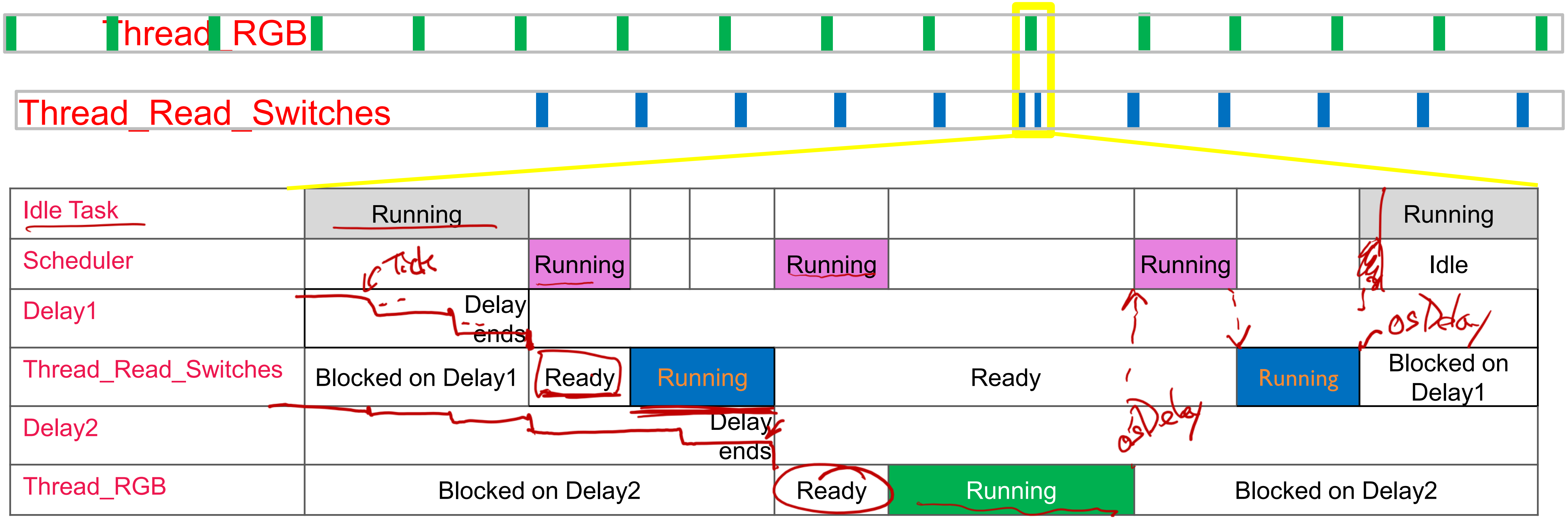


- Time sampling effects! Call to osDelay or osDelayUntil ***not synchronous*** with phase of timer events (overflows)
- Example: n = 2
  - **Block** until two ticks occur.
  - Minimum delay is >1 tick but <=2 ticks, depends on phase relation
- Then thread is marked as **ready** to run

- Summary: Two parts to time delay:
  - From **delay function call** to thread **becoming ready**
    - Based on timer phase and period
  - From thread **becoming ready** to when it **resumes running**
    - Based on which other processing is scheduled before this thread resumes

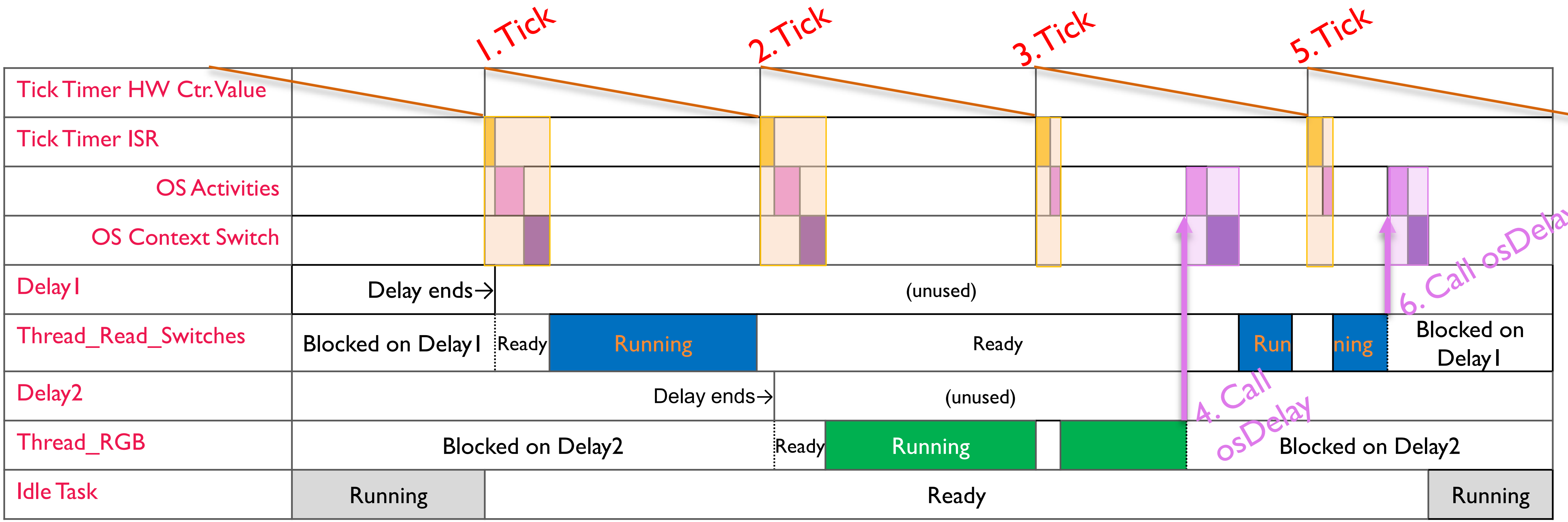
```
void MyThread (…) {
  while (1) {
        // Work
        Do_Some_Work();
        // Wait
        osDelay(2);
  }
}
```

Example Program Execution with Preemption: Zoom In



- Thread_Read_Switches runs after Delay1 expires
- When Delay2 expires, scheduler sees Thread_RGB is higher priority than Thread_Read_Switches, so swaps them
- When Thread_RGB blocks (waiting for next Delay2), scheduler resumes running Thread_Read_Switches
- Thread_Read_Switches runs until it blocks (waiting for Delay1)

Example Program Execution with Preemption: Extreme Zoom



1. Timer Tick Interrupt
2. Timer Tick Interrupt
3. Timer Tick Interrupt
4. Thread_RGB calls osDelay
5. Timer Tick Interrupt
6. Thread_Read_Switches calls osDelay

osDelayUntil Example

https://arm-software.github.io/CMSIS_5/RTOS2/html/group__CMSIS__RTOS__Wait.html

```
#include "cmsis_os2.h"
void Thread_1 (void *arg) {// Thread function
  uint32_t tick;
  tick = osKernelGetTickCount();  // retrieve the number of system ticks
  for (;;) {
    tick += 1000U;                        // delay 1000 ticks periodically
    osDelayUntil(tick);
    // ...
  }
}
```

- Allows creation of periodic delays which don’t accumulate error from delayed thread runs

- Delay until a specific tick
- Each tick is numbered, so can get number of current tick with osKernelGetTickCount()