Embedded System Technologies for Deep Learning and Approximate Computing

Under the SPARC Project P:271 – "Approximate Computing Techniques for Resource Constrained Edge Devices"

Prof. Alessandro Cilardo
acilardo@unina.it

Day 2
May 30th, 2021

Microcontroller-based systems: software development

Cortex-M software development flow and tools

- Numerous vendors selling C compiler suites for Cortex-M microcontrollers
- For example:
  - Keil^{TM} Microcontroller Development Kit (MDK-ARM)
  - ARM DS-5^{TM} (Development Studio 5)
  - GNU Compiler Collection (GCC)
  - IAR Systems (Embedded Workbench for ARM Cortex-M)
  - Red Suite from Code Red Technologies (acquired by NXP in 2013)
  - Mentor Graphics Sourcery CodeBench
  - mbed.org, free open source IoT OS and development tools
  - Atollic TrueStudio (STM32Cube)

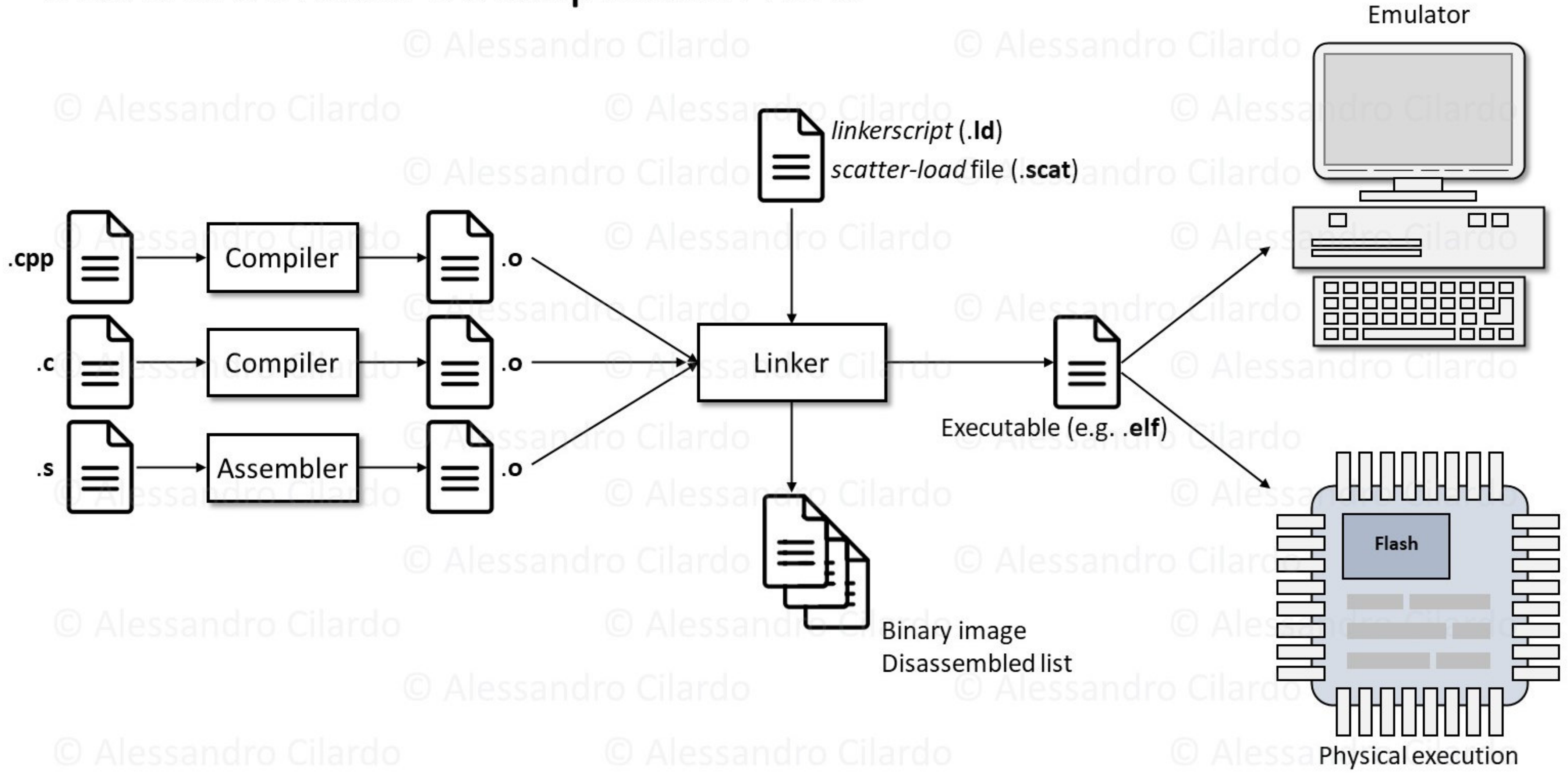
Typical software development tools

- C compiler
  - translates *C* program files into object files
- Assembler
  - translates *assembly* code files into object files
- Linker
  - joins multiple object files together and defines memory configuration
- Flash programmer
  - transfers the compiled program image to the flash memory of the microcontroller
- Debugger
  - controls the operation of the microcontroller and accesses internal information so that status of the system can be examined and the program operations can be checked
- Simulator
  - allows the program execution to be simulated without real hardware
- Other utilities
  - various tools, for example to convert the compiled files into different formats

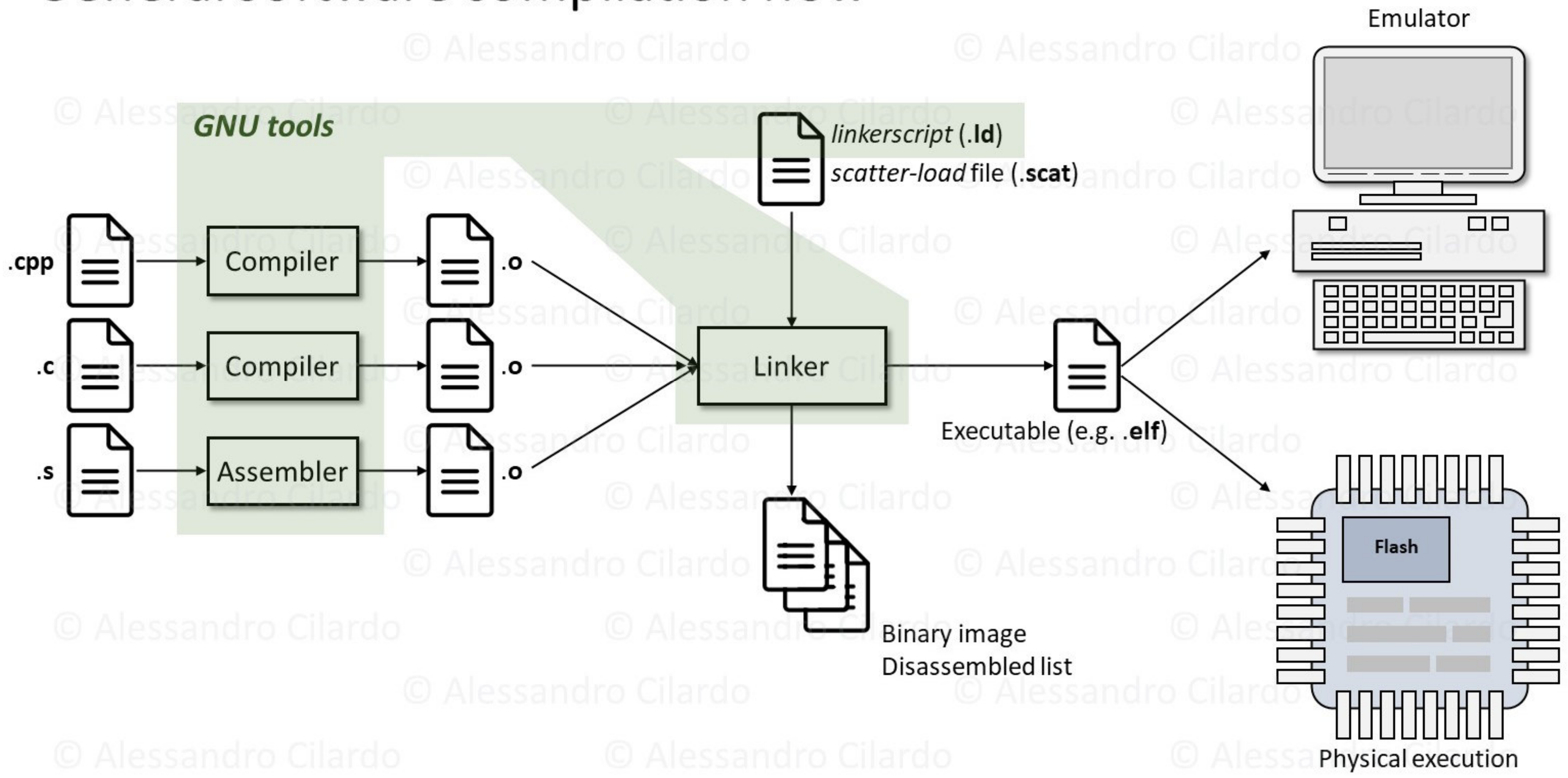
What does an embedded software dev flow need to care about?

- The hardware architecture is more or less directly exposed to the software
- Many aspects need to be explicitly addressed
  - access functions for configuring and accessing the peripheral registers
  - system configuration and management
  - interrupt setting
  - specifying the layout of the program and data memory
  - etc..
- The flow is often dependent on the specific vendor and tools
  - microcontroller vendors usually provide specific header files and C codes
  - aspects like memory layout can be specified in a tool-specific manner
  - e.g. with "scatter-loading" files in some toolchains
  - or, through command line options to specify the locations of ROM and RAM
  - in a GNU-based toolchain, the memory specification is handled by linker scripts

General software compilation flow

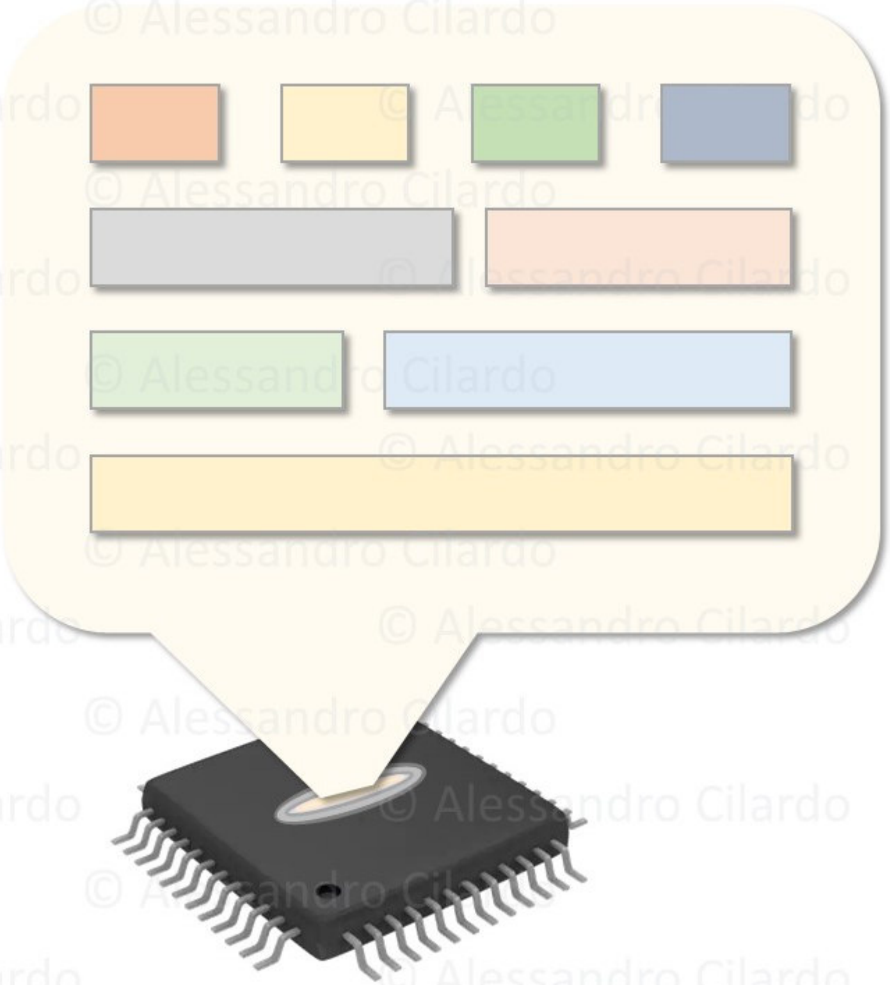


General software compilation flow



Types of embedded software

- Software developed by in house developers
- Software reused from other projects
- Device-driver libraries from microcontroller vendors
- Embedded Operating Systems
- Other third-party software products such as communication protocol stacks
- . . .



Cortex-M low-level software: how are peripherals accessed?

- Registers are memory mapped: Write to memory (peripherals blocks) through pointers
- For example, consider a GPIO device:

```
// "GPIO A Port" Configuration Register Low
#define GPIOA_CRL (*((volatile unsigned long *) (0x40010800)))
// "GPIO A Port" Configuration Register High
#define GPIOA_CRH (*((volatile unsigned long *) (0x40010804)))
. . .
void GPIOA_reset(void){
    GPIOA_CRL = 0; // Bits 0 to 7 correspond to pins, all set as analog input mode
. . .
```

- Problems:
  - the approach is not scalable
  - many addresses to store (many registers, to be distinguished for each peripheral)
  - difficult to write general functions working for multiple instances of the same peripheral type

Low-level software: how are peripherals accessed?

- Define the peripheral registers as data structures:

```
typedef struct{
__IO uint32_t CRL;
__IO uint32_t CRH;
__IO uint32_t IDR;
__IO uint32_t ODR;
__IO uint32_t BSRR;
__IO uint32_t BRR;
__IO uint32_t LCKR;
} GPIO_TypeDef;
```

- Then each peripheral base address (GPIO A to GPIO G) is defined as a pointer to the data structure:

```
// Peripheral base address (in the bit-band region)
#define PERIPH_BASE ((uint32_t)0x40000000)
. . .
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
#define GPIOA_BASE (APB2PERIPH_BASE + 0x0800)
. . .
#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
. . .
```

Low-level software: how are peripherals accessed?

- Note: **__IO** is defined in a standardized header file in CMSIS
  - It implies a volatile data item (e.g., a peripheral register), which can be read or written to by software
  - A peripheral register can also be defined as "**__I**" (read only) and "**__O**" (write only)

```
#ifdef __cplusplus
#define __I volatile // for read-only operations
#else
#define __I volatile const // for read-only operations
#endif
#define __O volatile  // for write-only operations
#define __IO volatile // for read/write operations
```

Low-level software: how are peripherals accessed?

- By using structures, we can create functions that can be invoked for each instance of the peripheral easily
- For example, the code to reset the GPIO port can be written as:

```
void GPIO_reset(GPIO_TypeDef* GPIOx){
    GPIOx->CRL = 0; // Bit 0 to 7 set as analog input
    GPIOx->CRH = 0; // Bit 8 to 15 set as analog input
    GPIOx->ODR = 0; // Default output value will be 0
    return;
}
```

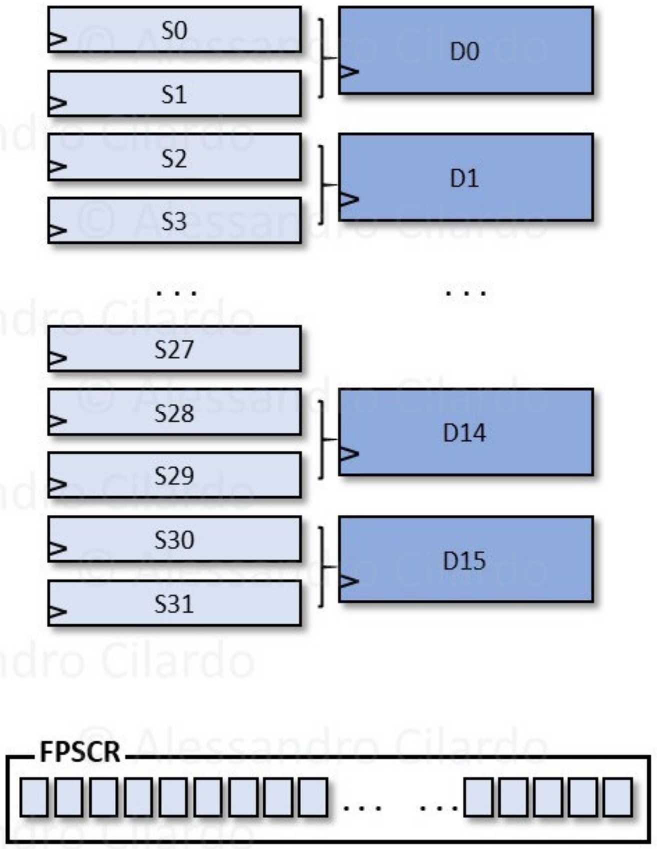
- To use this function, we just need to pass the peripheral base pointer to the function:

```
GPIO_reset(GPIOA); // Reset GPIO A
GPIO_reset(GPIOB); // Reset GPIO B
. . .
```

- This method for declaring peripheral registers is used by almost all of the Cortex-M microcontroller device-driver packages

Cortex-M low-level vector programming (VFP)

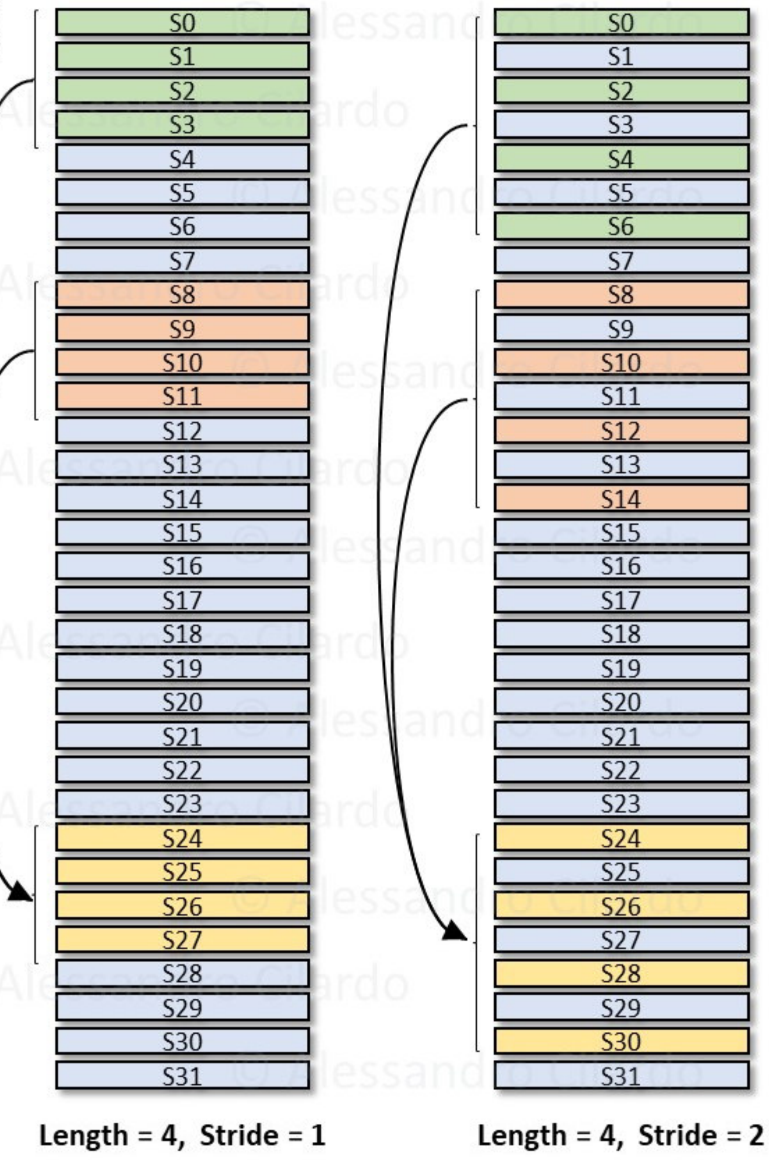
- Single Instruction Multiple Data (SIMD) instructions and associated SIMD registers
  - (only Single-Precision in Cortex-M4, SP/DP in M7)
- Floating-point registers are accessed in *banks* for SIMD operations
  - e.g. the [`S4`, `S6`, `S8`, `S10`] set is a four-element bank
- Two parameters (length, ***L***, stride, ***S***) are set in the Floating-Point Status/Control Register (FPSCR)
  - a SIMD bank will be formed as ***L*** registers spaced by ***S*** positions in the register file
  - all floating point operations following a write to FPSCR will be performed in SIMD fashion if ***L***>1

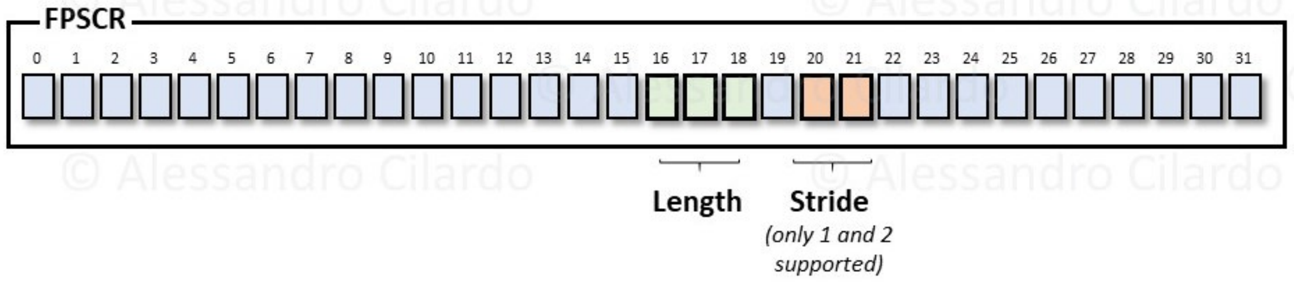


Cortex-M low-level vector programming

- a couple of examples of SIMD floating point addition operations
  - ***Length***=4 elements processed by a single instruction
  - first example: ***Stride***=1 → consecutive registers form a bank
  - second example: ***Stride***=2 →registers in a bank are spaced by 2 indices
- The same instruction **`fadd`** can be executed affecting different numbers and positions of registers, depending on ***Length*** and ***Stride***

```
fadds s24, s0, s8
```





Cortex-M low-level vector programming (VFP)

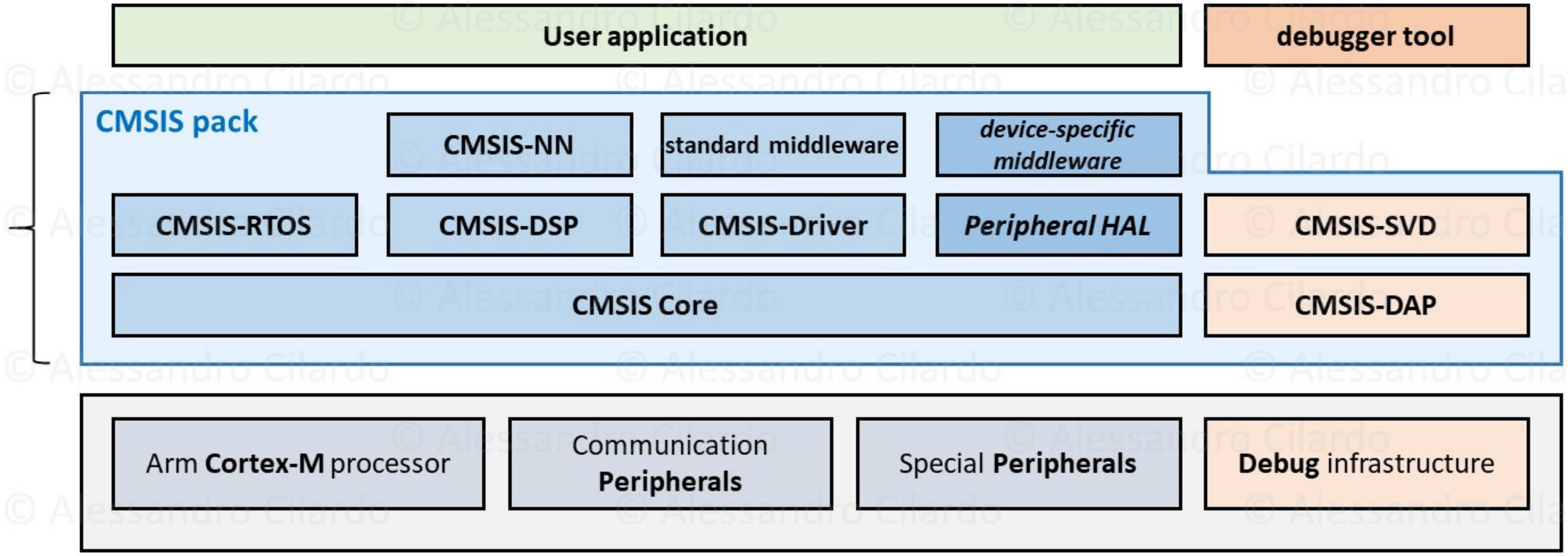
- Two parameters (length, ***L***, stride, ***S***) are set in the FP control register FPSCR
  - ***L*** registers spaced by ***S*** are accessed by a single SIMD instructions
- An example: perform element-wise addition between two vectors

```
lbl_1:                        r0 contains the configuration value for FPSCR
    fmxr fpscr, r0            the stride field in FPSCR is set to 1, length to 8
                              r3 contains the size of the array
    asr r7, #3                then, divide r3 by 8 (assume it's a multiple of 8)

.loop:
    fldmias r1!,{s8-s15}      load 8 floats from the first source array
    fldmias r2!,{s16-s23}     load 8 floats from the second source array
    fadds s24, s8, s16        perform s[24..31] = s[8..15] + s[16..23]
    fstmias r5!,{s24-s31}     store the result in the destination array
    subs r7, r7, #1           update the loop counter
    bne .loop
    . . .
    fmxr fpscr, r9            the length field is set back to 8 to clear SIMD mode
```

Cortex Microcontroller Software Interface Standard (CMSIS)

- Developed by ARM with various microcontroller vendors, tools vendors, and software solution providers. Ensures:
  - Enhanced software reusability, Enhanced software compatibility, Easy C-based development, Toolchain independence, Open-source access



CMSIS (as of v5.7.0)

Cortex Microcontroller Software Interface Standard *(as of v5.7.0)*

- Core(M) (all Cortex-M, SecurCore)
  - Standardized API for the Cortex-M core and peripherals. Includes intrinsic functions for Cortex-M4/M7/M33/M35P SIMD instructions.
- Core(A) (Cortex-A5/A7/A9)
  - Standardized API and basic run-time system for the Cortex-A5/A7/A9 processor core and peripherals.
- Driver (all Cortex)
  - Generic peripheral driver interfaces for middleware (e.g communication stacks, file systems, or graphic user interfaces)
- DSP (all Cortex-M)
  - 60+ functions for fixed-point (fractional q7, q15, q31) and SP-FP (32-bit). SIMD versions available for Cortex-M4/M7/M33/M35P
- NN (all Cortex-M)
  - Efficient neural network kernels maximizing the performance and minimizing the memory footprint on Cortex-M processor cores.
- RTOS v1 (Cortex-M0/M0+/M3/M4/M7)
  - Common API for RTOSs along with a reference implementation based on RTX. It enables portability across multiple RTOSs.
- RTOS v2 (all Cortex-M, Cortex-A5/A7/A9
  - Extends CMSIS-RTOS v1 with Armv8-M support, dynamic object creation, provisions for multi-core systems, binary compatible interface.
- Pack (all Cortex-M, SecurCore, Cortex-A5/A7/A9
  - Delivery mechanisms for software components, device parameters, and evaluation board support.
- Build (all Cortex-M, SecurCore, Cortex-A5/A7/A9)
  - A set of tools, software frameworks, and work flows that improve productivity, for example with Continuous Integration (CI).
- SVD (all Cortex-M, SecurCore)
  - Peripheral description of a device that can be used to create peripheral awareness in debuggers or CMSIS-Core header files.
- DAP (all Cortex)
  - Firmware for a debug unit that interfaces to the CoreSight Debug Access Port.
- Zone (all Cortex-M)
  - Defines methods to describe system resources and to partition these resources into multiple projects and execution areas.

CMSIS: most important components

- CMSIS-DSP library
  - released in 2010, supports many common DSP operations such as FFT and filters, allowing software developers to create DSP applications on Cortex-M microcontrollers easily
- CMSIS-SVD (System View Description)
  - an XML-based file format to describe peripheral set in microcontroller products. Debug tool vendors can use the SVD files from microcontroller vendors to construct peripheral viewers
- CMSIS-RTOS
  - an API specification for embedded OS running on Cortex-M, allowing reusability and portability, as middleware/applications can be developed for multiple embedded OSs
- CMSIS-DAP (Debug Access Port)
  - a reference design for a debug interface adaptor, supporting USB to JTAG/Serial conversions, allowing low-cost debug adaptors to work for multiple toolchains

CMSIS-Core (Cortex-M processor support)

- A set of APIs for application or middleware developers to access the features on the Cortex-M processor
  - regardless of the microcontroller devices or toolchain used
- implements the basic run-time system for a Cortex-M device
- gives the user access to the processor core and the device peripherals
- defines:
  - Hardware Abstraction Layer (HAL) for system peripherals
  - support to system exceptions without compatibility issues
  - Naming conventions for device-specific interrupts and header file organization for improved software portability
  - Methods for system initialization
  - Intrinsic functions for CPU instructions that are not supported by standard C
  - A variable for the system clock frequency which simplifies the setup of timers
  - etc..

CMSIS-Core: details

- Standardized definitions for the processor's peripherals
  - include the registers in the Nested Vector Interrupt Controller (NVIC), a system tick timer in the processor (SysTick), an optional Memory Protection Unit (MPU), various programmable registers in the System Control Block (SCB)
- Standardized access functions to access processor's features
  - include various functions for interrupt control using NVIC, and functions for accessing special registers in the processors.
- Standardized functions for accessing special instructions easily
  - The Cortex-M processors support a number of instructions for special purposes (e.g., Wait-For-Interrupt, **`Wfi`**, for entering sleep mode), instead of relying on the toolchain, e.g. for intrinsic functions or inline assembly
- Standardized function names for system exception handlers
  - A number of system exception types are presented in the architecture for the Cortex-M processors. By giving the corresponding system exception handlers standardized names, it makes it much easier to develop software solutions that can be applied to multiple Cortex-M products. This is especially important for embedded OS developers.
- Standardized functions for system initialization
  - control configuration of clock circuitry and power management registers before the application starts. In CMSIS, these steps are placed in a function called **`SystemInit()`**. The actual implementation of this function is device specific and might need adaption for various project requirements, but the standardized naming and location simplifies development
- Standardized software variables for clock speed information
  - knowing the actual clock frequency might be needed for setting up the baud rate divider in a UART, or to initialize the SysTick timer for an embedded OS. A software variable called **`SystemCoreClock`** is made available for that
- A common platform for device-driver libraries
  - Each device-driver library has the same look and feel, making it easier for beginners to learn how to use the devices

CMSIS-Core

- CMSIS supports the complete range of Cortex-M processors and the Armv8-M/v8.1-M architecture including security extensions:
  - Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, Cortex-M7
  - Cortex-M23, Cortex-M33
- CMSIS also supports the following Cortex-M processor variants:
  - Cortex-M1
    - a processor designed specifically for implementation in FPGAs (Armv6-M architecture)
  - SecurCore SC000
    - designed specifically for smartcard and security applications (Armv6-M architecture)
  - SecurCore SC300
    - designed specifically for smartcard and security applications (Armv7-M architecture)
  - Cortex-M35P
    - a temper resistant Cortex-M processor with optional software isolation using TrustZone for Armv8-M

CMSIS coding rules

- The CMSIS uses the following essential coding rules and conventions:
  - Compliant with ANSI C (C99) and C++ (C++03)
  - Uses ANSI C standard data types defined in <**`stdint.h`**>
  - Variables and parameters have a complete data type
  - Expressions for **`#define`** constants are enclosed in parenthesis
  - Conforms to MISRA 2012 (but does not claim MISRA compliance). MISRA rule violations are documented
- In addition, the CMSIS recommends the following conventions for identifiers:
  - CAPITAL names to identify Core Registers, Peripheral Registers, and CPU Instructions
  - CamelCase names to identify function names and interrupt functions
  - *namespace_* prefixes avoid clashes with user identifiers and provide functional groups (i.e. for peripherals, RTOS, or DSP Library)
- The CMSIS is documented within the source files with:
  - Comments that use the C or C++ style
  - Doxygen compliant function comments that provide:
    - brief function overview
    - detailed description of the function
    - detailed parameter explanation
    - detailed information about return values

CMSIS validation

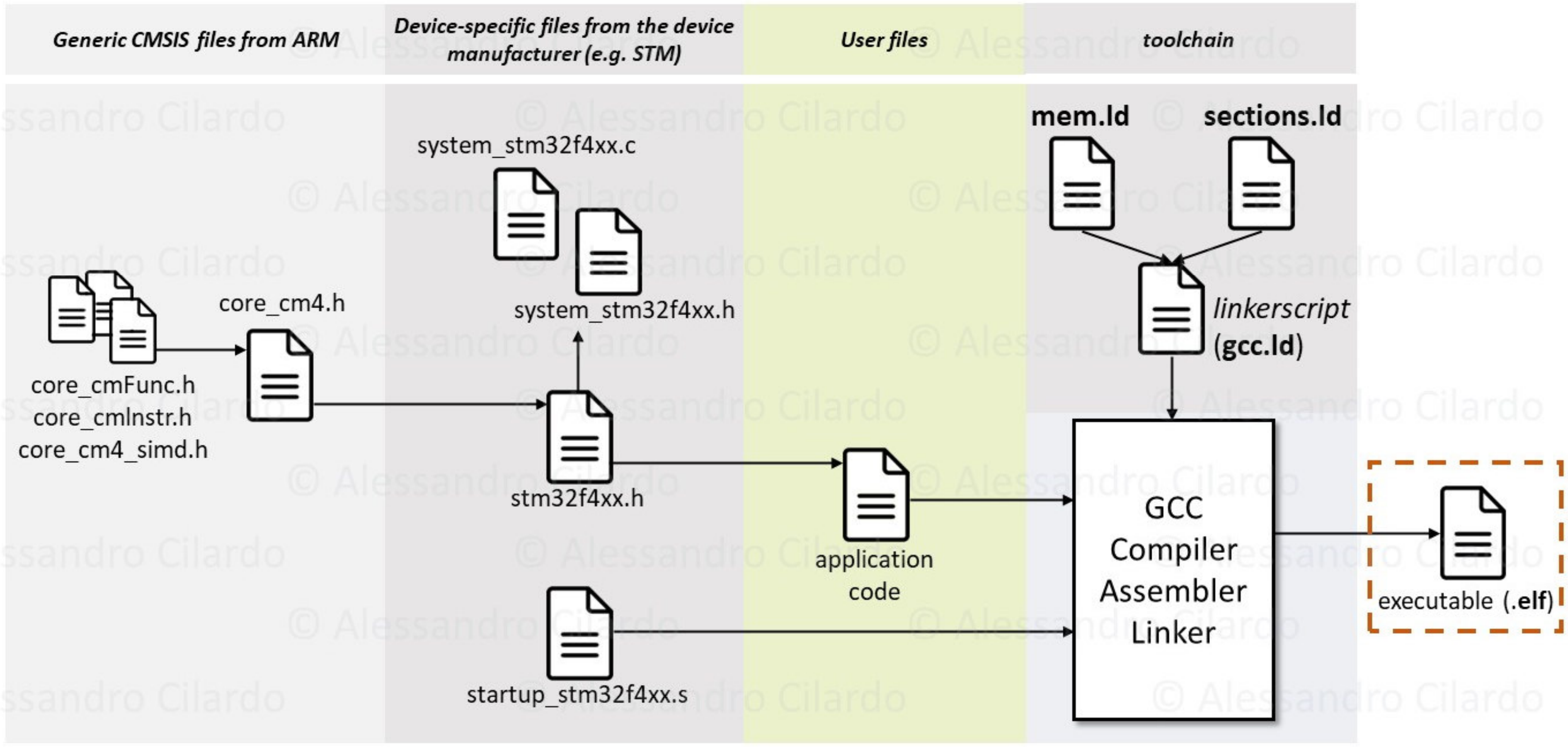
- Components of CMSIS Version 5 validated using mainstream compilers
  - to get a diverse coverage, Arm uses the Arm Compiler v5 (based on EDG front-end), the Arm Compiler v6 (based on LLVM front-end), and the GCC Compiler in the various tests
- CMSIS components are compatible with a range of C and C++ language standards
  - comply with the Application Binary Interface (ABI) for the Arm Architecture (exception CMSIS-RTOS v1) →C API interfaces ensuring inter-operation between various toolchains
- The scope of the run-time test coverage is limited
  - CMSIS API interfaces and functions need to scale to many processors and devices
  - however, several components are validated using dedicated test suites
- CMSIS source code checked for MISRA C:2012 conformance using PC-Lint
  - deviations are documented with reasonable effort
  - however Arm does not claim MISRA compliance
  - note: CMSIS source code not checked for MISRA C++:2008 conformance (may be incompatible with C standards, e.g. for warnings generated by the various C compilers)

GNU Compiler Collection (gcc) toolchain

- The gcc toolchain contains a C compiler, assembler, linker, libraries, debugger, and additional utilities
  - e.g. **`arm-none-eabi-as`** is the Arm-targeted assembler tool
  - (*note*: the tool is pre-built for ARM EABI 3 *without* any specific target OS platform, hence the prefix "**`none`**." Some GNU toolchains could be created for developing applications for Linux platforms, and in those cases the prefix would be "**`arm-linux-`**")

Tools	Generic command name	Command Name in GNU Tools for ARM Embedded Processors
C compiler	**`gcc`**	**`arm-none-eabi-gcc`**
assembler	**`as`**	**`arm-none-eabi-as`**
linker	**`ld`**	**`arm-none-eabi-ld`**
Binary file generation tool	**`objcopy`**	**`arm-none-eabi-objcopy`**
Disassembler	**`objdump`**	**`arm-none-eabi-objdump`**

Example of a project structure using CMSIS-core



Example of a project structure using CMSIS-core

mem.ld defines the memory map (flash and SRAM) of the microcontroller:

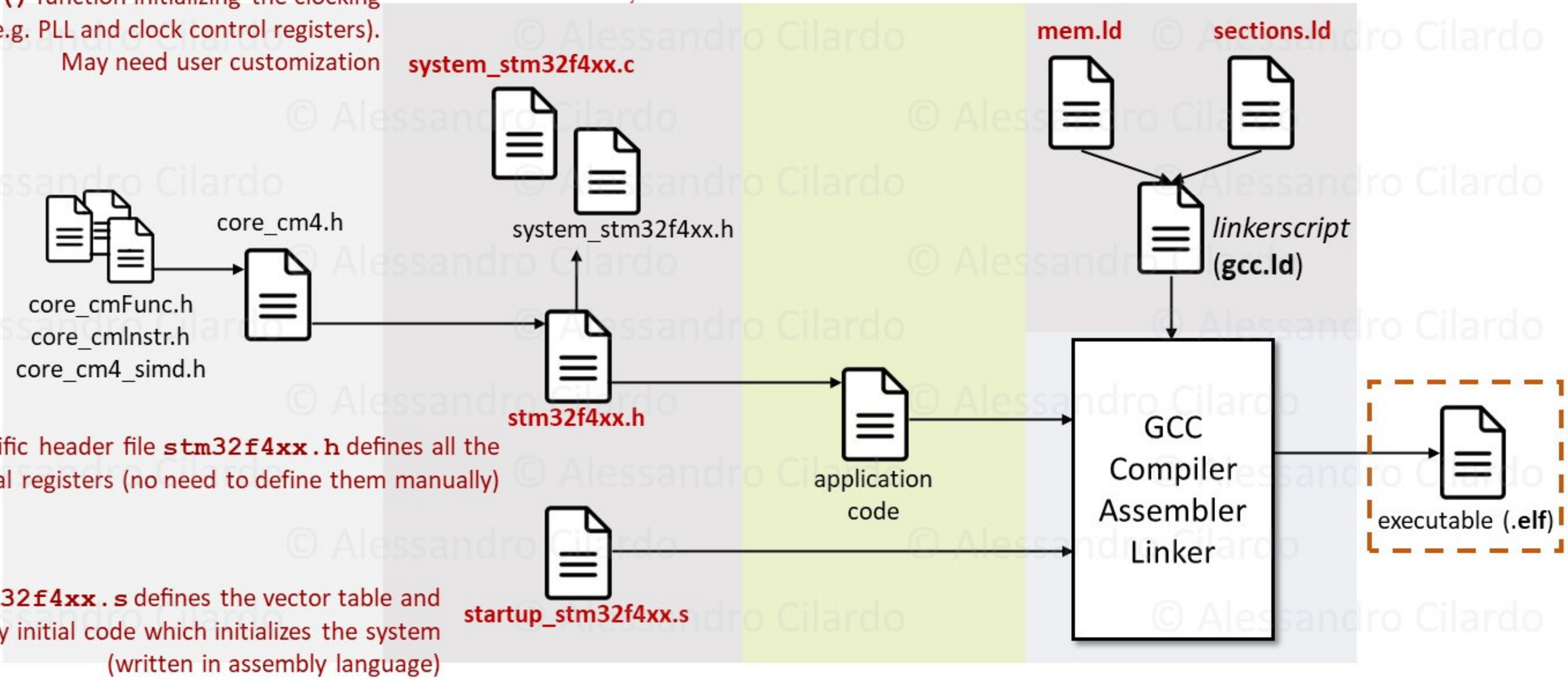
```
MEMORY
{
FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 1024K
RAM (xrw)  : ORIGIN = 0x20000000, LENGTH = 112K
}
```

sections.ld defines the layout inside the executable image (included in the GNU Tools, can be used as is)

The `system_stm32f4xx.c` provides the `SystemInit()` function initializing the clocking system (e.g. PLL and clock control registers). May need user customization

Device-specific header file `stm32f4xx.h` defines all the peripheral registers (no need to define them manually)

`startup_stm32f4xx.s` defines the vector table and contains the very initial code which initializes the system (written in assembly language)



ARM / STM32 dev tools and libraries: a few useful links

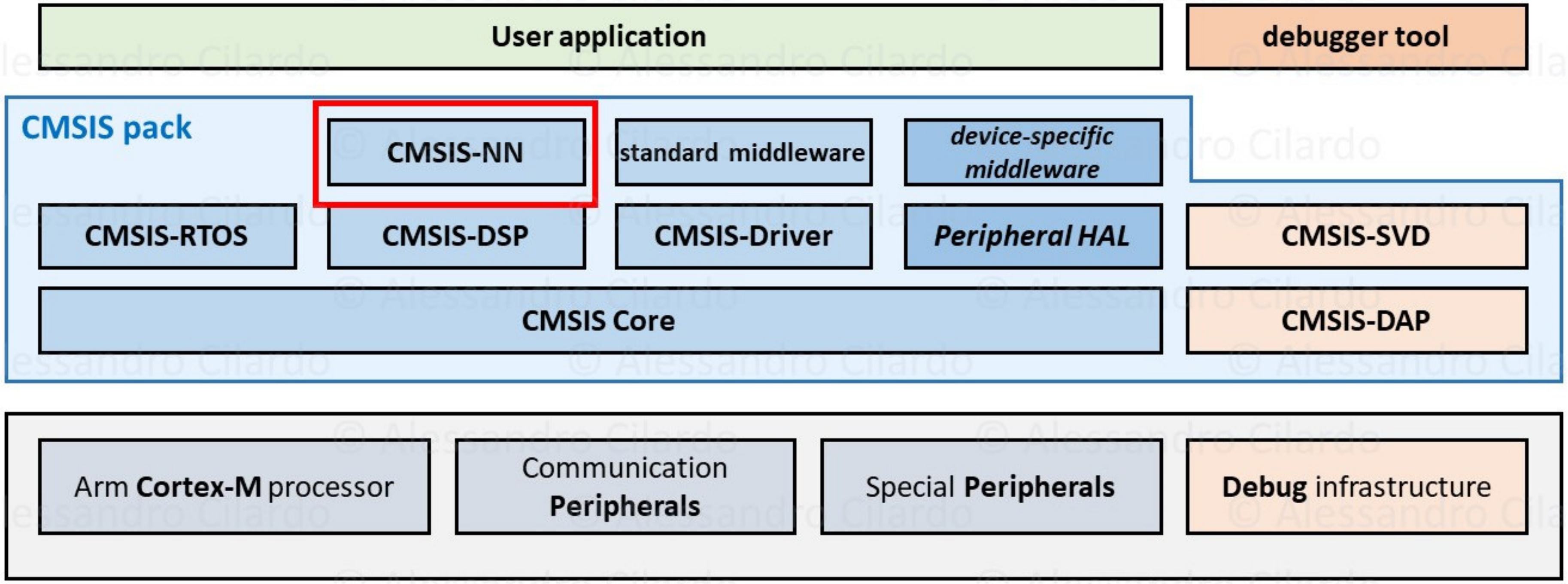
- GNU Arm Embedded Toolchain:
  - `https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads`
- *For Windows users*:

  Windows Build Tools (as a lightweight alternative to MSYS2 or CygWin):
  - `https://github.com/gnu-mcu-eclipse/windows-build-tools/releases`
- CMSIS v.5:
  - `https://github.com/ARM-software/CMSIS_5`
- xPack OpenOCD:
  - `https://github.com/xpack-dev-tools/openocd-xpack/releases`
- STM32CubeF4 MCU Firmware Package (*includes CMSIS*)
  - `https://github.com/STMicroelectronics/STM32CubeF4`

CMSIS-DSP software library

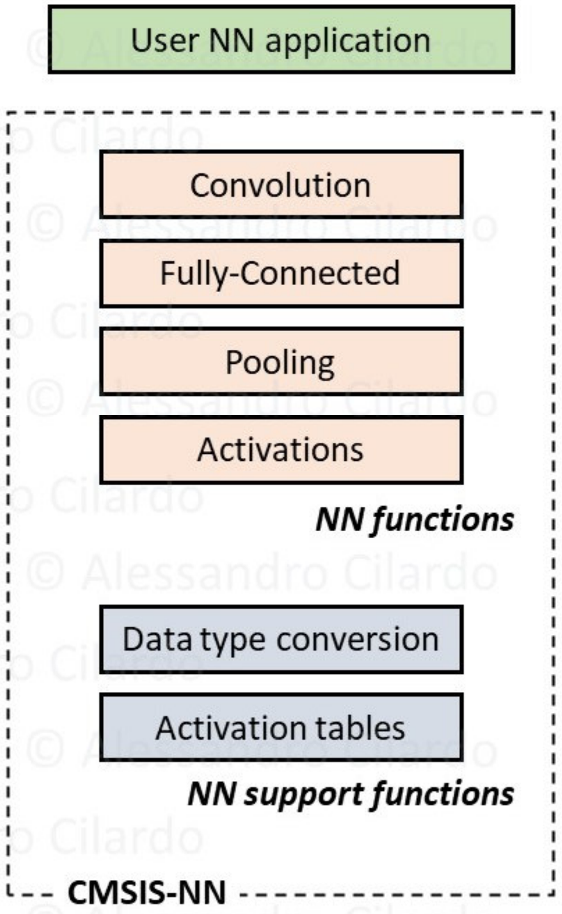
- A suite of common signal processing functions for use on Cortex-M and -A processors
- Provides a number of functions, covering:
  - Basic math functions
  - Fast math functions
  - Complex math functions
  - Filtering functions
  - Matrix functions
  - Transform functions
  - Motor control functions
  - Statistical functions
  - Support functions
  - Interpolation functions
  - Support Vector Machine functions (SVM)
  - Bayes classifier functions
  - Distance functions
- In most cases, provides separate functions for:
  - 8-bit, 16-bit, 32-bit integer values
  - 32-bit floating-point values

RTOS support in ARM CMSIS and the STM32 Cube platform



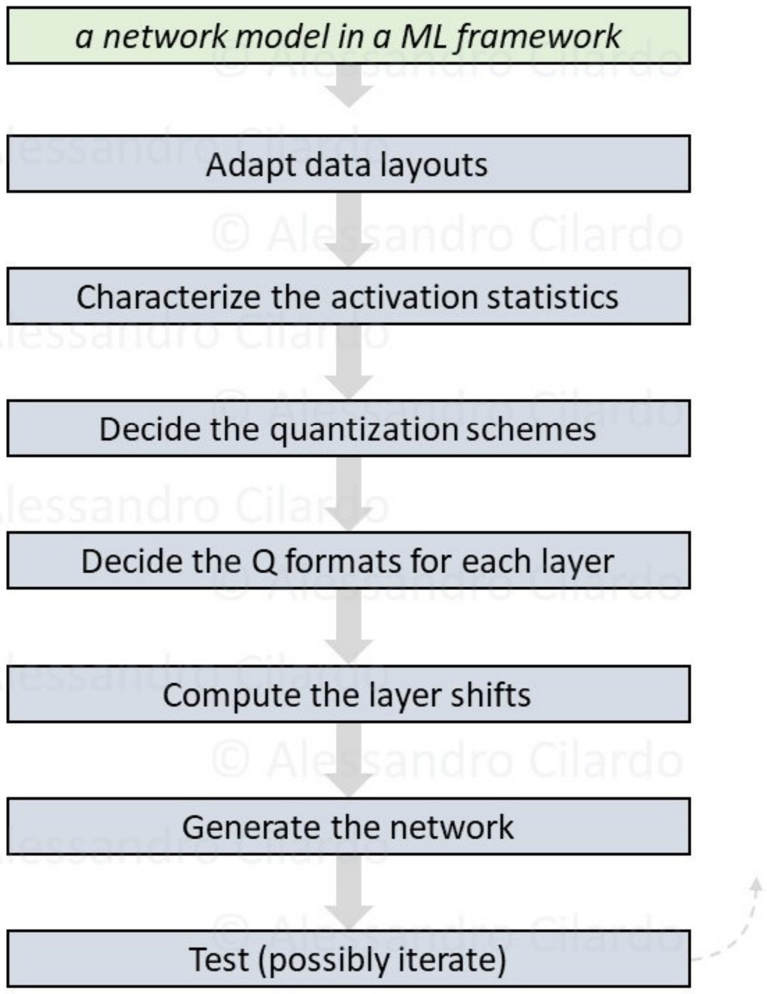
CMSIS-NN software library

- A collection of efficient neural network kernels for Cortex-M processors
- Maximizes performance while minimizing the memory footprint
- Provides a number of functions, covering:
  - Convolution Functions
  - Activation Functions
  - Fully-connected Layer Functions
  - Pooling Functions
  - Softmax Functions
  - Basic math Functions
- Provides separate functions for operating on different weight and activation data types, including
  - 8-bit integers (q7_t)
  - 16-bit integers (q15_t)
- The functions can be classified into two segments
  - Legacy functions supporting ARM's internal symmetric quantization (8 bits), identified with their suffix of _q7 or _q15 (*now discontinued*)
  - Functions that support TensorFlow Lite framework with symmetric quantization (8 bits), identified by the _s8 suffix and can be invoked from TFL micro (*the latter types of functions are bit exact to TensorFlow Lite*)



CMSIS-NN software library: converting a network for Cortex-M

- CMSIS-NN supports a limited number of layers
  - for unsupported layers: need to manually combine CMSIS-NN layers an functions from CMSIS-DSP
- Weights to be reordered if:
  - The layer is a convolutional layer or a fully connected layer
  - The input of the layer is a matrix or tensor
  - The ML framework and the CMSIS-NN orderings are different
- Quantization
  - the CMSIS-NN flow supports only quantization of existing FP networks (no training of quantized networks)
  - Compute activation statistics
  - Decide the quantization schemes
  - Determine the Q-formats (8- and 16-bit supported). Some constraints may apply for certain layers
- Compute layer shifts
- Generate the network
- Test (possibly, iterate the process)

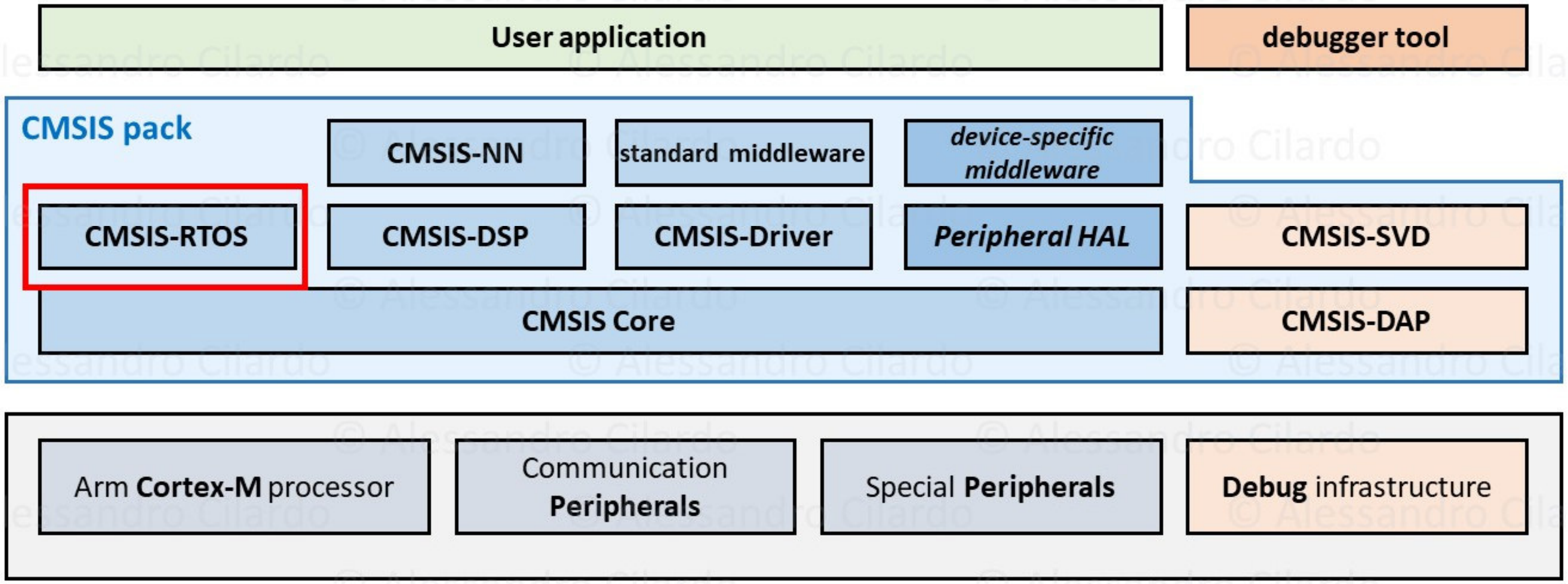

a network model in a ML framework
Adapt data layouts
Characterize the activation statistics
Decide the quantization schemes
Decide the Q formats for each layer
Compute the layer shifts
Generate the network
Test (possibly iterate)

CMSIS-NN software library: design and evaluation (*)

- Quantization: 32-bit FP → *q7*, *q15* Q-format (benefits performance and footprint, still *ok* for inference)
  - developed kernels supporting both 8-bit and 16-bit data
  - wise data transformation, e.g. from q7_t to q15_t using *SIMD within a register* (SWAR)
- Matrix Multiplication, based on the *mat_mult* kernels in CMSIS, 2x2 kernel
  - one batch, small kernel. The limited number of registers can be a challenge when implementing larger kernels
  - reorder the matrix weights so that row data are interleaved and can be read with only one pointer access
- Convolution
  - to contain footprint, only expand a limited number of columns (e.g. 2), still *ok* to get the maximum performance
  - Height-Width-Channel (HWC) layout (pixels are consecutive, efficient use of SIMD)
- x-y pooling (faster than window-based pooling), based on *in-situ* destructive updates
  - 4.5 X speed-up, no memory overhead
- Activation function:
  - ReLu uses *SIMD within a register* (SWAR), Sigmoid and Tanh based on 256-entry tables and interpolation
- Tested for a CIFAR-10 dataset, q7_t quantization, an off-the-shelf Arm Cortex-M7 platform
  - results: 87 kB footprint for weights, 55 kB footprint for activations
  - classify **10.1** images per second with an accuracy of **79.9%** (vs. **80.3%** achieved by the pre-quantized network)
  - **4.6X** Throughput, **4.9X** Energy efficiency *vs*: 1D convolution (*arm_conv* from CMSIS-DSP) + Caffe-like pooling + ReLU

(*) L. Lai, N. Suda, V. Chandra, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs", 2018

RTOS support in ARM CMSIS and the STM32 Cube platform



Real-Time Operating System (RTOS): FreeRTOS

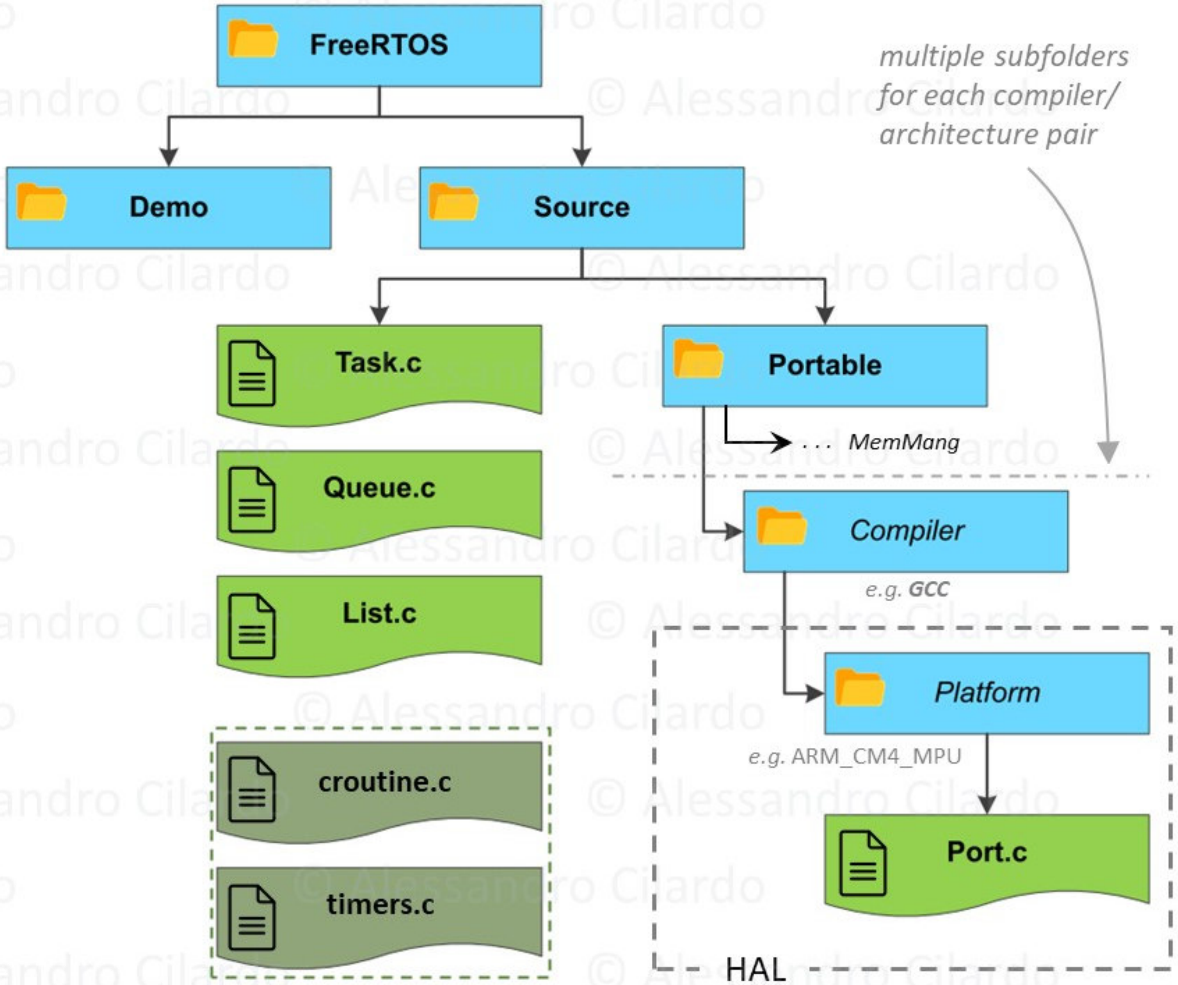
- FreeRTOS™ is an RTOS designed to fit in a microcontroller
  - deeply embedded applications
  - provides the core real-time scheduling functionality, inter-task communication, timing and synchronization primitives only
  - could more appropriately be defined as *real-time kernel* or *real-time "executive"*
  - additional functionality (command console interface, networking stacks) can be included with add-on components
- FreeRTOS in the STM platform relies on the common APIs provided by the CMSIS-RTOS wrapping layer
  - applications using FreeRTOS can be directly ported on any other RTOS
  - no need to modify the high level APIs, only the CMSIS-OS wrapper has to be changed

Overview of FreeRTOS

- Support for 27 architectures, including ARM7, Arm Cortex-M0, Cortex-M3, Cortex-M4
- FreeRTOS-MPU supports the Arm® Cortex®-M3 Memory Protection Unit
- Small footprint, e.g. 4K to 9K bytes
- Very portable code structure predominantly written in C
- Supports both tasks and co-routines
- Queues, binary semaphores, counting semaphores, recursive semaphores and mutexes for communication and synchronization between tasks, or between tasks and interrupts
- Mutexes with priority inheritance
- Supports efficient software timers
- Powerful execution traces functionality
- Stack overflows detection options
- No software restriction on the number of tasks that can be created or priorities that can be used
- No restrictions imposed on priority assignment (more than one task can be assigned the same priority)
- Royalty free
- Free development tools and embedded software source code

FreeRTOS source organization

- The real-time kernel is contained in just four files
  - software timers and co-routine functionality require additional files
- Source files available for every processor port, and for every demonstration application
- The portable folder also includes several sample heap allocation schemes in the MemMang subfolder (`heap_x.c`)
- `FreeRTOSConfig.h` controls most settings and configuration choices



FreeRTOS API

Task creation
xTaskCreate
vTaskDelete

Task control
vTaskDelay
vTaskDelayUntil
uxTaskPriorityGet
vTaskPrioritySet
vTaskSuspend
vTaskResume
xTaskResumeFromISR
vTaskSetApplicationTag
xTaskCallApplicationTaskHook

Task utilities
xTaskGetCurrentTaskHandle
xTaskGetSchedulerState
uxTaskGetNumberOfTasks
vTaskList
vTaskStartTrace
ulTaskEndTrace
vTaskGetRunTimeStats

Kernel control
vTaskStartScheduler
vTaskEndScheduler
vTaskSuspendAll
xTaskResumeAll

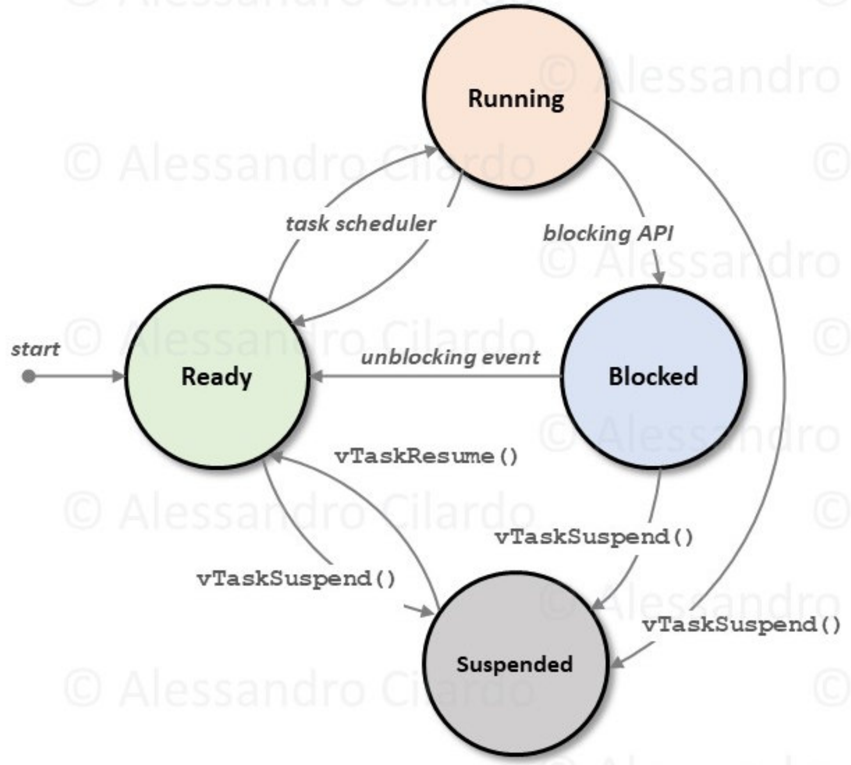
Queue management
xQueueCreate
xQueueSend
xQueueReceive
xQueuePeek
xQueueSendFromISR
xQueueSendToBackFromISR
xQueueSendToFrontFromISR
xQueueReceiveFromISR
vQueueAddToRegistry
vQueueUnregisterQueue
xQueueCreate

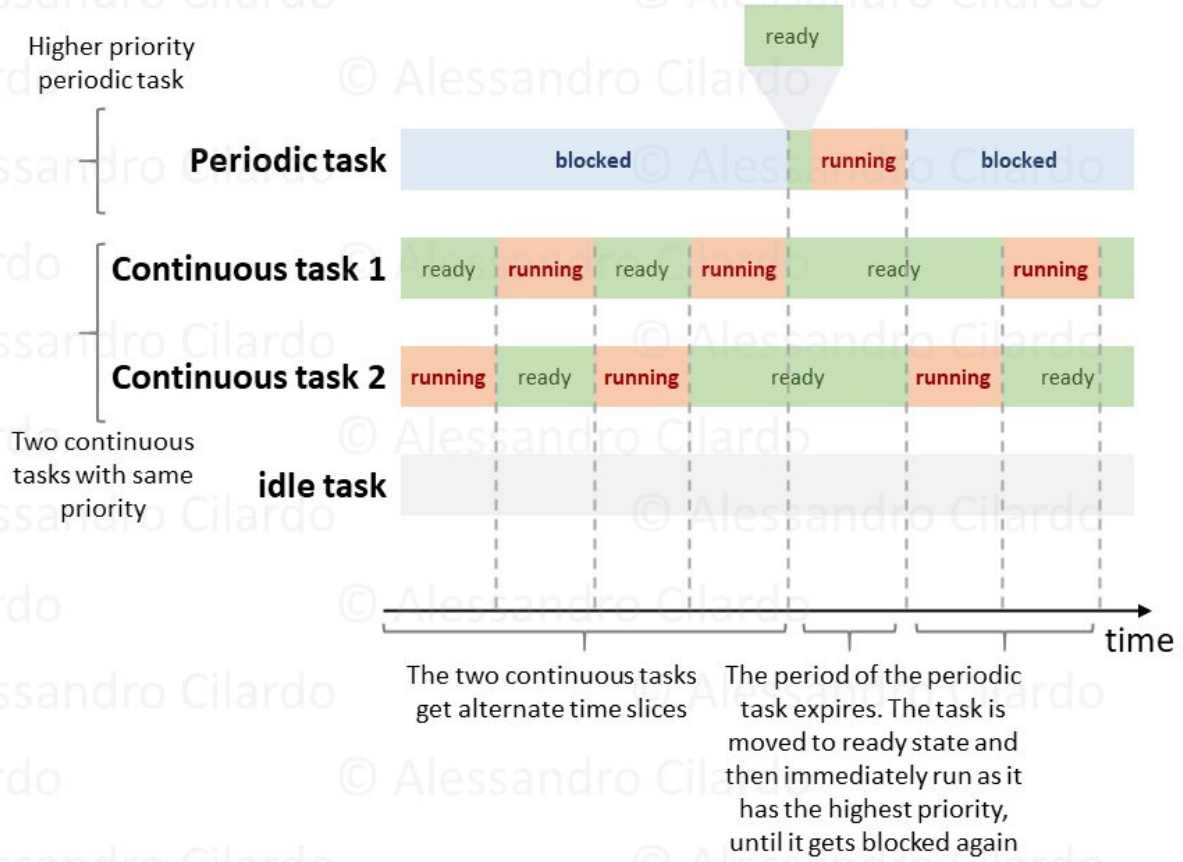
Semaphores
vSemaphoreCreateBinary
vSemaphoreCreateCounting
xSemaphoreCreateMutex
xSemaphoreTake
xSemaphoreGive
xSemaphoreGiveFromISR

FreeRTOS memory management

- static vs. dynamic
  - Static optionally used (since v9.0.0) for: Tasks, Software Timers, Queues, Event Groups, Binary Semaphores, Counting Semaphores, Recursive Semaphores, Mutexes
  - static allocation avantages: controlled addresses and footprint, no allocation checks needed
- Dynamic memory management (heap):
  - **heap_1**: the very simplest, does not permit memory to be freed
  - **heap_2**: permits memory to be freed, but does not coalescence adjacent free blocks
  - **heap_3**: simply wraps the standard `malloc()` and `free()` supplied with the user's compiler for thread safety
  - **heap_4**: coalescences adjacent free blocks to avoid fragmentation. Includes absolute address placement option
  - **heap_5**: as per heap_4, with the ability to span the heap across multiple non-adjacent memory areas

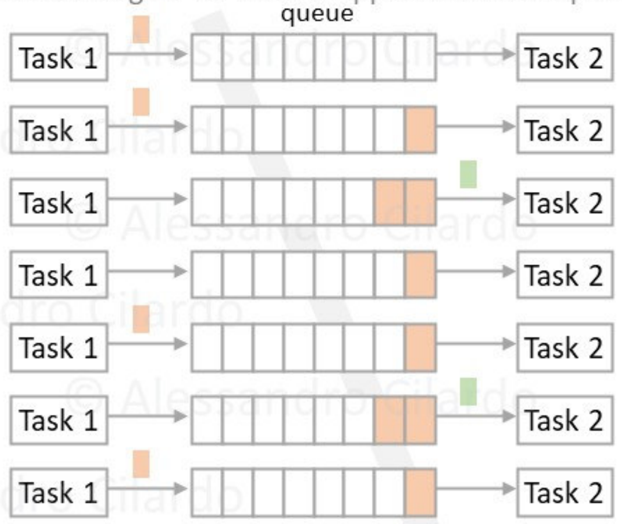
FreeRTOS task management

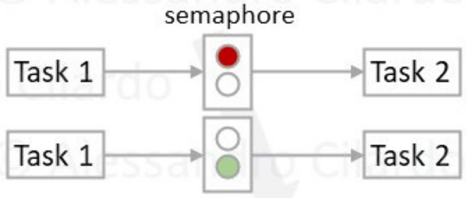


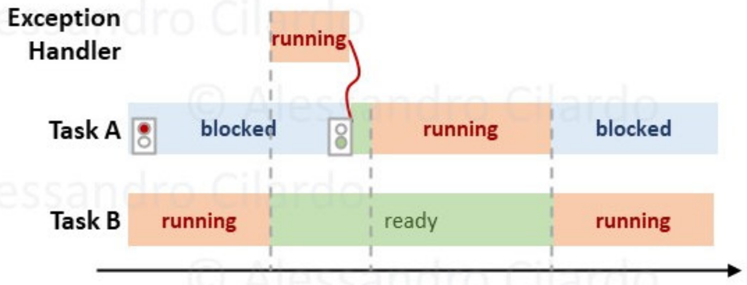

Higher priority periodic task
Periodic task
blocked
ready
running
blocked
Continuous task 1
ready
running
ready
running
ready
running
Continuous task 2
running
ready
running
ready
running
ready
Two continuous tasks with same priority
idle task
time
The two continuous tasks get alternate time slices
The period of the periodic task expires. The task is moved to ready state and then immediately run as it has the highest priority, until it gets blocked again

FreeRTOS: further topics

- Queue management
- Mailboxes, binary and counting semaphores, mutexes
- Software timer management
- Interrupt management
  - *note*: re-entrant routines denoted with ..`FromISR(..)`
  - no need to determine the context (from Task vs. from ISR) in the API code
  - e.g. interrupt safe version of the Binary Semaphore API can be used to unblock a task each time a particular interrupt occurs
- Resource management
- Event groups
- Task notification
- ...



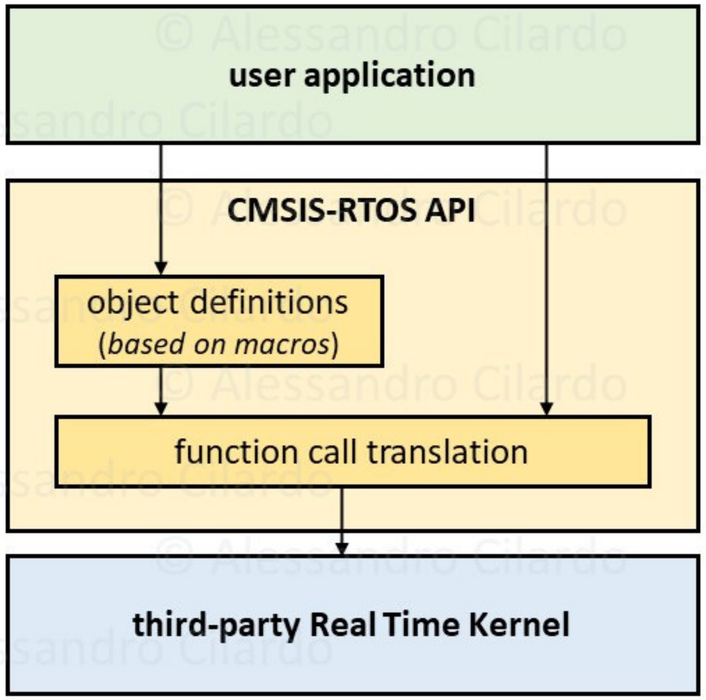




Reference: Richard Barry, *Mastering the FreeRTOS™ Real Time Kernel - A Hands-On Tutorial Guide*

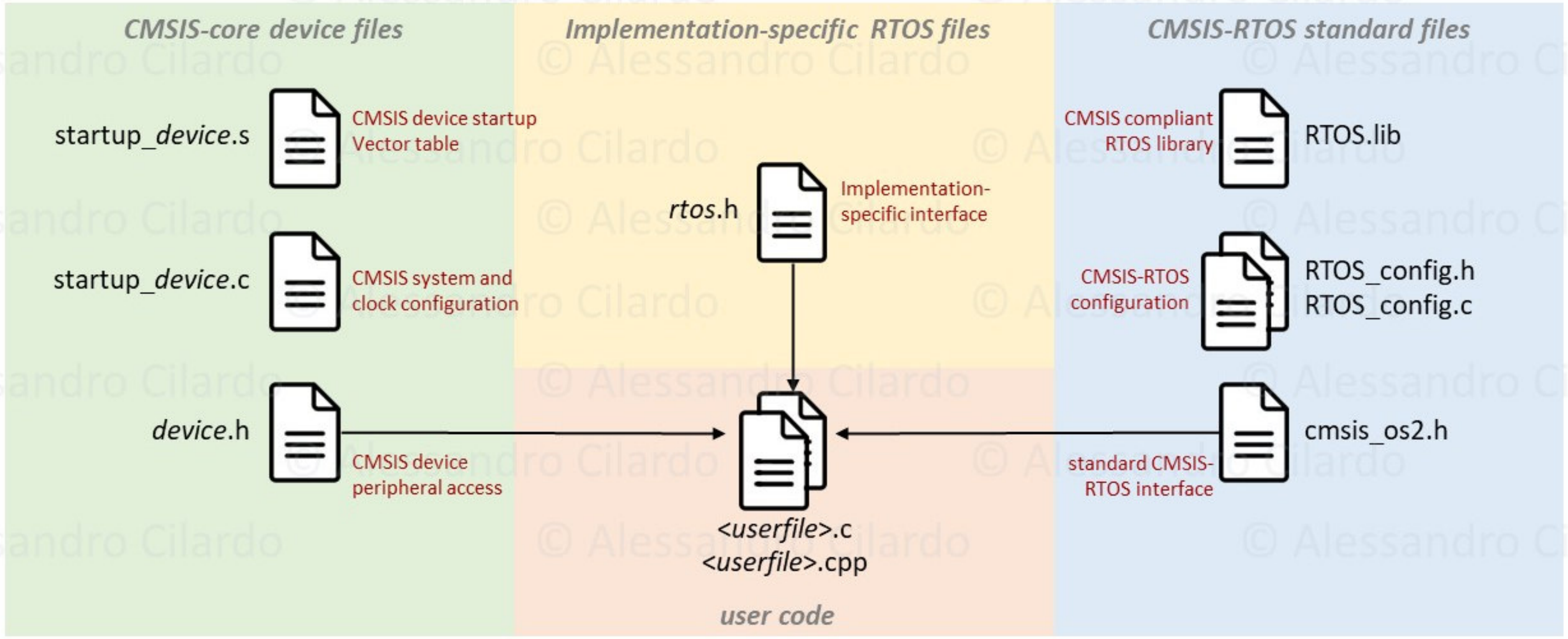
CMSIS-RTOS (v2)

- A common API for Real-Time operating systems
  - standardized programming interface that is portable to many RTOS
  - enables software templates, middleware, libraries, and other components that can work across the supported RTOS systems
- What it provides:
  - intuitive naming for functions, parameters, ..
  - high-level abstraction for thread management
  - portable interface for scheduling functionality (**`osDelay`**, **`osWait`**, **`osThreadYield`**,..)
  - most functions can be called from ISR
  - communication support between threads and/or ISRs (signals, messages, mails)
  - Mutex and semaphore management
- Kernel objects defined and accessed using macros
  - allows kernels to be optimized for specific processors
- A few optional features include:
  - Support for multi-processor systems
  - Support for Cortex-M Memory Protection Unit (MPU) and DMA controller
  - Zero-copy mail queue
  - Deterministic or round-robin context switching
  - Deadlock avoidance, for example with priority inversion
  - Zero interrupt latency using Cortex instructions **`LDEX`** and **`STEX`**



CMSIS-RTOS2 File Structure

- CMSIS-based applications are extended for RTOS by adding a CMSIS-RTOS2 component
  - such a component may be provided as library or source code (library case shown below)
  - the `cmsis_os2.h` header file gives access to RTOS API functions and is the only interface header required when dynamic object allocation is used
- Static object allocation requires access to RTOS object control block definitions
  - An implementation specific header file (`rtos.h`) provides access to such definitions


CMSIS-core device files
Implementation-specific RTOS files
CMSIS-RTOS standard files
startup_*device*.s
CMSIS device startup Vector table
startup_*device*.c
CMSIS system and clock configuration
device.h
CMSIS device peripheral access
rtos.h
Implementation-specific interface
CMSIS compliant RTOS library
RTOS.lib
CMSIS-RTOS configuration
RTOS_config.h
RTOS_config.c
cmsis_os2.h
standard CMSIS-RTOS interface
<userfile>.c
<userfile>.cpp
user code

CMSIS-RTOS2 example of code

```c
// CMSIS-RTOS 'main' function template
#include "RTE_Components.h"
#include  CMSIS_device_header
#include "cmsis_os2.h"

// Application main thread
void app_main (void *argument) {

  // ...
  for (;;) {}
}

int main (void) {

  // System Initialization
  SystemCoreClockUpdate();
  // ...

  osKernelInitialize();                 // Initialize CMSIS-RTOS
  osThreadNew(app_main, NULL, NULL);    // Create application main thread
  osKernelStart();                      // Start thread execution
  for (;;) {}
}
```

CMSIS-RTOS2 API

- Kernel Information and Control
  - osKernelGetInfo : Get RTOS Kernel Information.
  - osKernelGetState : Get the current RTOS Kernel state.
  - osKernelGetSysTimerCount : Get the RTOS kernel system timer count.
  - osKernelGetSysTimerFreq : Get the RTOS kernel system timer frequency.
  - osKernelInitialize : Initialize the RTOS Kernel.
  - osKernelLock : Lock the RTOS Kernel scheduler.
  - osKernelUnlock : Unlock the RTOS Kernel scheduler.
  - osKernelRestoreLock : Restore the RTOS Kernel scheduler lock state.
  - osKernelResume : Resume the RTOS Kernel scheduler.
  - osKernelStart : Start the RTOS Kernel scheduler.
  - osKernelSuspend : Suspend the RTOS Kernel scheduler.
  - osKernelGetTickCount : Get the RTOS kernel tick count.
  - osKernelGetTickFreq : Get the RTOS kernel tick frequency.
- Thread Management
  - osThreadDetach : Detach a thread (thread storage can be reclaimed when thread terminates).
  - osThreadEnumerate : Enumerate active threads.
  - osThreadExit : Terminate execution of current running thread.
  - osThreadGetCount : Get number of active threads.
  - osThreadGetId : Return the thread ID of the current running thread.
  - osThreadGetName : Get name of a thread.
  - osThreadGetPriority : Get current priority of a thread.
  - osThreadGetStackSize : Get stack size of a thread.
  - osThreadGetStackSpace : Get available stack space of a thread based on stack watermark recording during execution.
  - osThreadGetState : Get current thread state of a thread.
  - osThreadJoin : Wait for specified thread to terminate.
  - osThreadNew : Create a thread and add it to Active Threads.
  - osThreadResume : Resume execution of a thread.
  - osThreadSetPriority : Change priority of a thread.
  - osThreadSuspend : Suspend execution of a thread.
  - osThreadTerminate : Terminate execution of a thread.
  - osThreadYield : Pass control to next thread that is in state READY.
- Thread Flags
  - osThreadFlagsSet : Set the specified Thread Flags of a thread.
  - osThreadFlagsClear : Clear the specified Thread Flags of current running thread.
  - osThreadFlagsGet : Get the current Thread Flags of current running thread.
  - osThreadFlagsWait : Wait for one or more Thread Flags of the current running thread to become signaled.
- Event Flags
  - osEventFlagsGetName : Get name of an Event Flags object.
  - osEventFlagsNew : Create and Initialize an Event Flags object.
  - osEventFlagsDelete : Delete an Event Flags object.
  - osEventFlagsSet : Set the specified Event Flags.
  - osEventFlagsClear : Clear the specified Event Flags.
  - osEventFlagsGet : Get the current Event Flags.
  - osEventFlagsWait : Wait for one or more Event Flags to become signaled.

CMSIS-RTOS2 API

- Generic Wait Functions
  - osDelay : Wait for Timeout (Time Delay).
  - osDelayUntil : Wait until specified time.
- Timer Management
  - osTimerDelete : Delete a timer.
  - osTimerGetName : Get name of a timer.
  - osTimerIsRunning : Check if a timer is running.
  - osTimerNew : Create and Initialize a timer.
  - osTimerStart : Start or restart a timer.
  - osTimerStop : Stop a timer.
- Mutex Management
  - osMutexAcquire : Acquire a Mutex or timeout if it is locked.
  - osMutexDelete : Delete a Mutex object.
  - osMutexGetName : Get name of a Mutex object.
  - osMutexGetOwner : Get Thread which owns a Mutex object.
  - osMutexNew : Create and Initialize a Mutex object.
  - osMutexRelease : Release a Mutex that was acquired by osMutexAcquire.
- Semaphores
  - osSemaphoreAcquire : Acquire a Semaphore token or timeout if no tokens are available.
  - osSemaphoreDelete : Delete a Semaphore object.
  - osSemaphoreGetCount : Get current Semaphore token count.
  - osSemaphoreGetName : Get name of a Semaphore object.
  - osSemaphoreNew : Create and Initialize a Semaphore object.
  - osSemaphoreRelease : Release a Semaphore token up to the initial maximum count.
- Memory Pool
  - osMemoryPoolAlloc : Allocate a memory block from a Memory Pool.
  - osMemoryPoolDelete : Delete a Memory Pool object.
  - osMemoryPoolFree : Return an allocated memory block back to a Memory Pool.
  - osMemoryPoolGetBlockSize : Get memory block size in a Memory Pool.
  - osMemoryPoolGetCapacity : Get maximum number of memory blocks in a Memory Pool.
  - osMemoryPoolGetCount : Get number of memory blocks used in a Memory Pool.
  - osMemoryPoolGetName : Get name of a Memory Pool object.
  - osMemoryPoolGetSpace : Get number of memory blocks available in a Memory Pool.
  - osMemoryPoolNew : Create and Initialize a Memory Pool object.
- Message Queue
  - osMessageQueueDelete : Delete a Message Queue object.
  - osMessageQueueGet : Get a Message from a Queue or timeout if Queue is empty.
  - osMessageQueueGetCapacity : Get maximum number of messages in a Message Queue.
  - osMessageQueueGetCount : Get number of queued messages in a Message Queue.
  - osMessageQueueGetMsgSize : Get maximum message size in a Memory Pool.
  - osMessageQueueGetName : Get name of a Message Queue object.
  - osMessageQueueGetSpace : Get number of available slots for messages in a Message Queue.
  - osMessageQueueNew : Create and Initialize a Message Queue object.
  - osMessageQueuePut : Put a Message into a Queue or timeout if Queue is full.
  - osMessageQueueReset : Reset a Message Queue to initial empty state.