



Everything You Need to Know about RTOSs in 30 Minutes

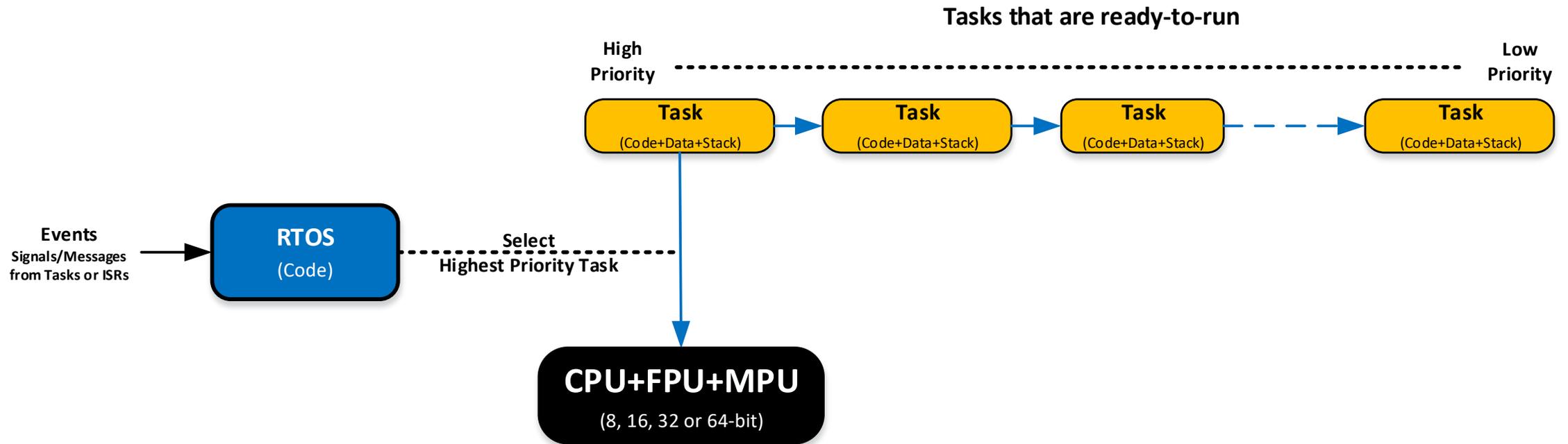
JEAN LABROSSE | DISTINGUISHED ENGINEER

EMBEDDED WORLD: FEBRUARY 26-28, 2019

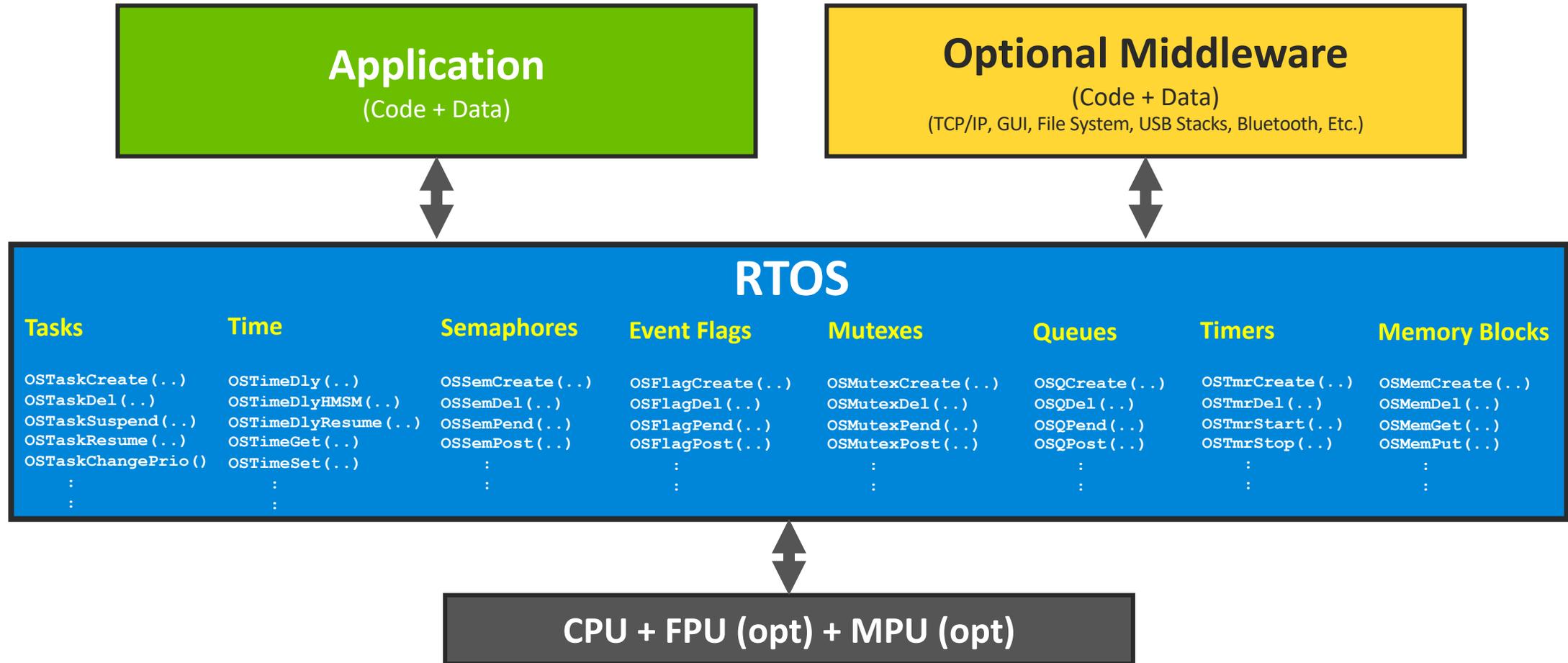


An RTOS Allows Multitasking

- An RTOS is software that manages the **time** and **resources** of a CPU
 - Application is split into **multiple tasks**
 - The RTOS's job is to **run the most important task** that is *ready-to-run*
 - On a single CPU, only **one task executes** at any given time

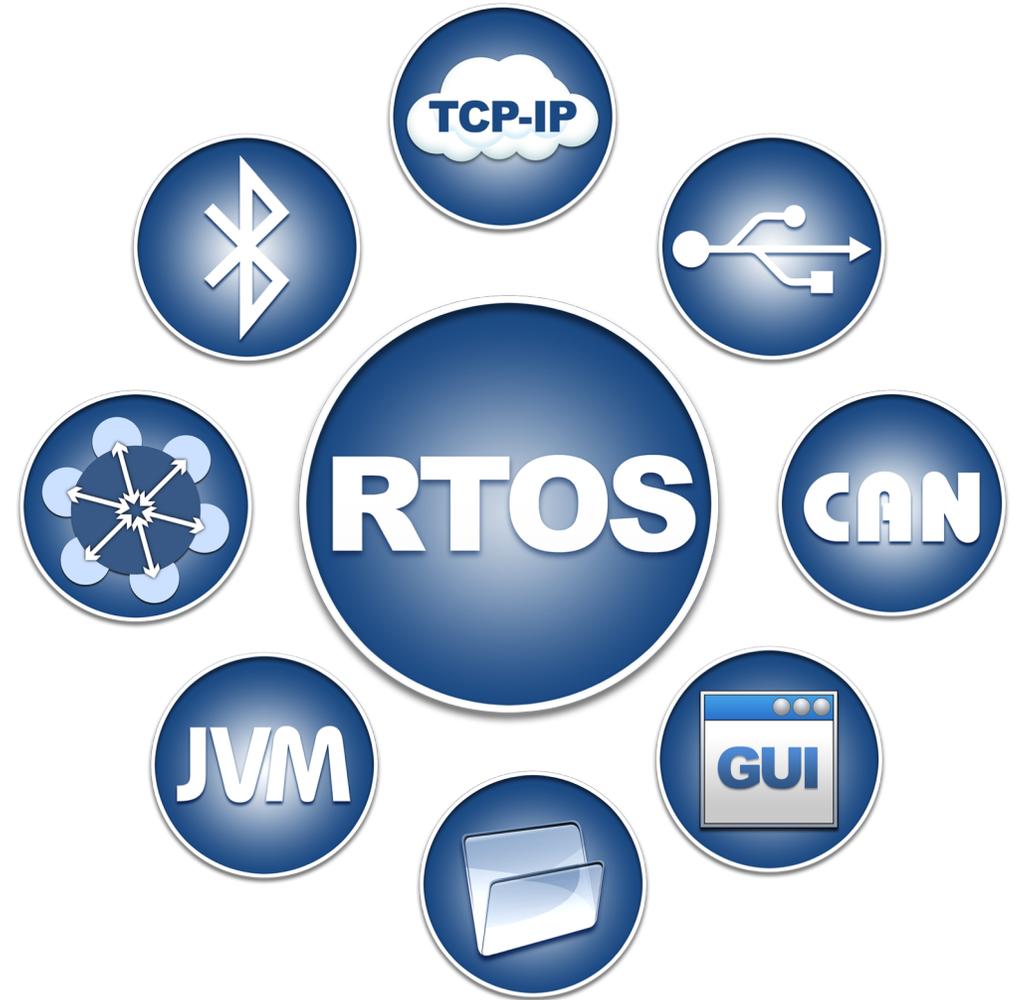


An RTOS Provides Services To Your Application



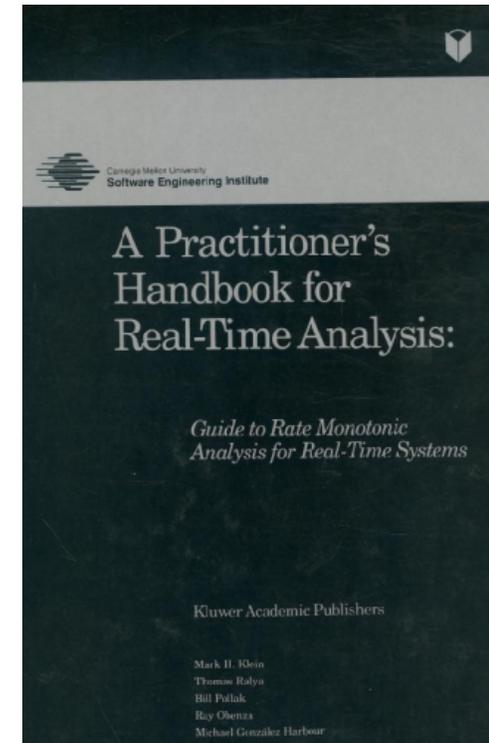
Benefits of Using an RTOS

- Allows you to **split** and **prioritize** the application code
 - The RTOS always runs the highest priority task that is ready
 - Adding low-priority tasks don't affect the responsiveness of high priority tasks
- Tasks **wait** for events
 - Avoids polling
- RTOSs make it **easy to add middleware** components
 - TCP/IP stack
 - USB stacks
 - File System
 - Graphical User Interface (GUI)
 - Etc.



Benefits of Using an RTOS

- Creates a **framework** for developing applications
 - Facilitate teams of multiple developers
- It's possible to **meet all the deadlines** of an application
 - Rate Monotonic Analysis (RMA) could be used to determine schedulability
- Most RTOSs have undergone **thorough testing**
 - Some are third-party certifiable, and even certified (DO-178B, IEC-61508, IEC-62304, etc.)
 - It's unlikely that you will find bugs in RTOSs
- RTOSs typically support many **different CPU architectures**
- Very easy to add **power management**



Drawbacks of Using an RTOS

- The RTOS itself is **code** and thus requires more Flash
 - Typically between 6-20K bytes
- An **RTOS requires extra RAM**
 - **Each task requires its own stack**
 - The size of each task depends on the application
 - **Each task** needs to be assigned a Task Control Block (TCB)
 - About 32 to 128 bytes of RAM
 - About 256 bytes for the RTOS variables
- **You** have to assign task priorities
 - Deciding on what priority to give tasks is not always trivial
- The services provided by the **RTOS consume CPU time**
 - Overhead is typically 2-10% of the CPU cycles
- There is a **learning curve** associated with the RTOS you select

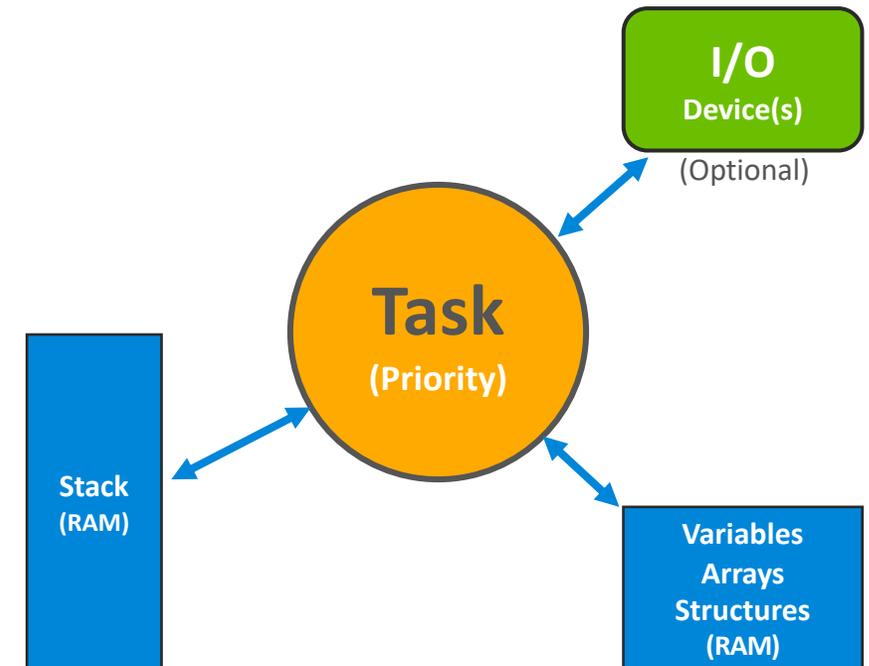
Tasks

- For each task:
 - **YOU** assign a **priority** based on its importance
 - Requires its own **Stack**
 - Manages its own variables, arrays and structures
 - Is typically an **infinite loop**
 - Possibly manages I/O devices
 - Contains **YOUR** application code

```
CPU_STK  MyTaskStk[MY_TASK_STK_SIZE]; // Task Stack

void MyTask (void *p_arg)           // Task Code
{
    Local Variables;

    Task initialization;
    while (1) {                     // Infinite Loop (Typ.)
        Wait for Event;
        Perform task operation;    // Do something useful
    }
}
```

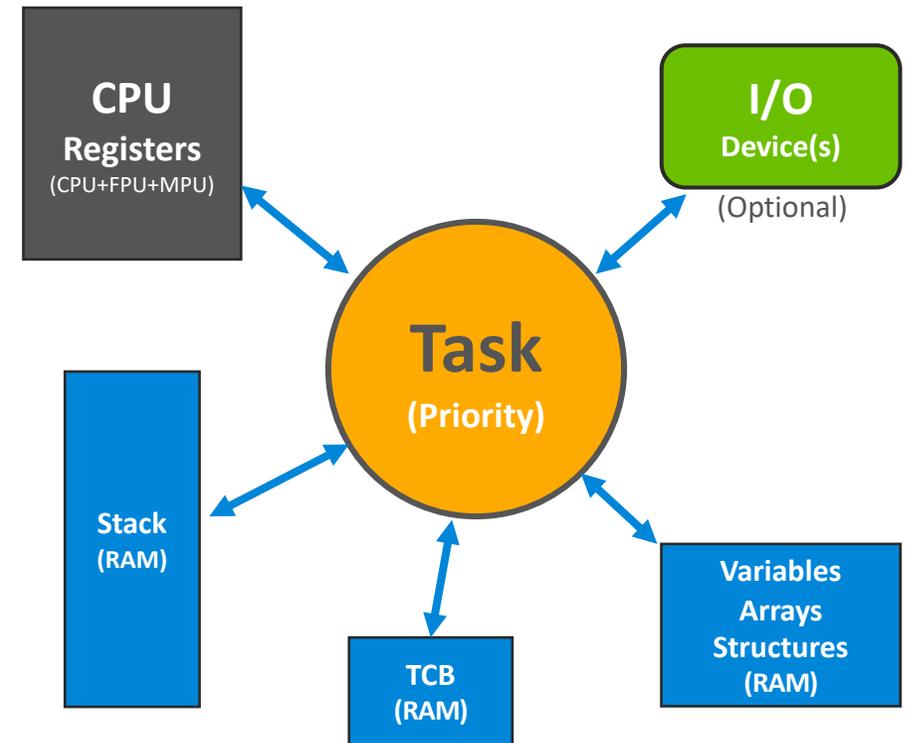


Creating a Task

- You must tell the RTOS about the existence of a task:
 - The RTOS provides a special API: `OSTaskCreate()` (or equivalent)

```
void OSTaskCreate (MyTask,           // Address of code
                  &MyTaskStk[0],    // Base of stack
                  MY_TASK_STK_SIZE,  // Size of stack
                  MY_TASK_PRIO,      // Task priority
                  :
                  :);
```

- The RTOS assigns the task:
 - Its own set of *CPU registers*
 - A Task Control Block (*TCB*)

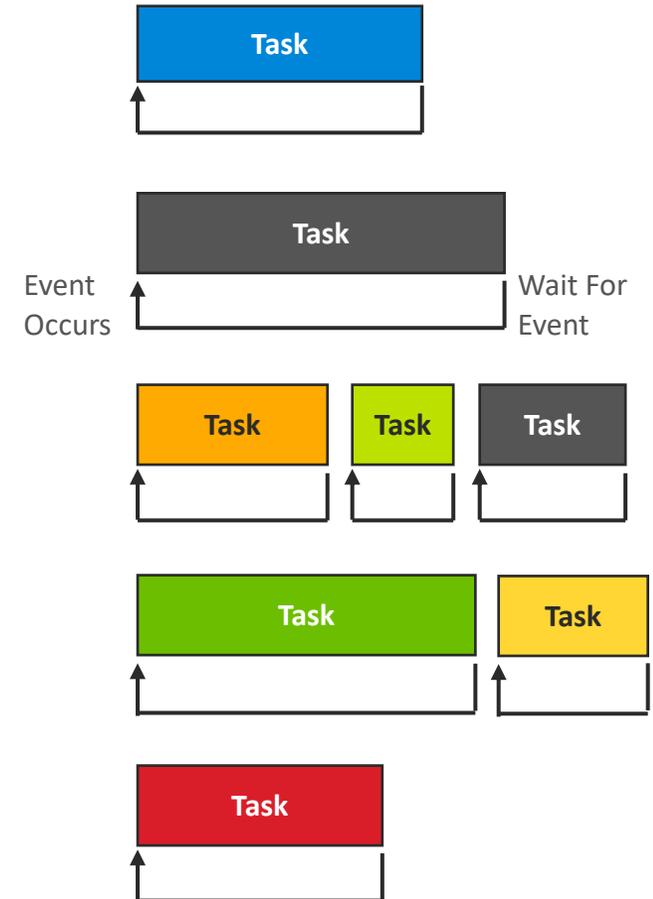


RTOSs Are Event Driven

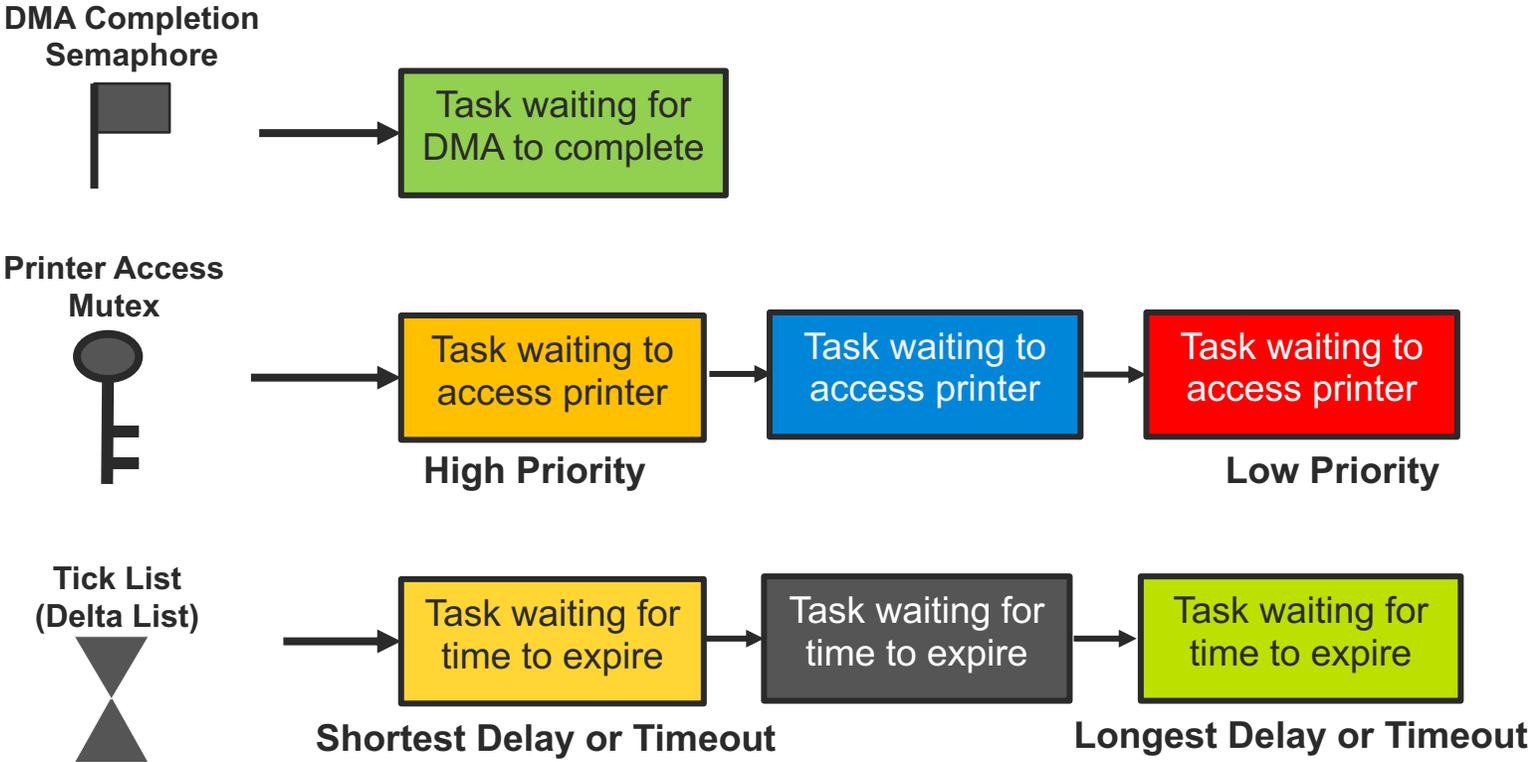
```
void EachTask (void)
{
  Task initialization;
  while (1) {
    Setup to wait for event;
    Wait for MY event to occur;
    Perform task operation;
  }
}
```

- **Only** the highest-priority Ready task can execute
 - Other tasks will run when the current task decides to **waits for its event**
- Ready tasks are placed in the RTOS's **Ready List**
- Tasks waiting for their event are placed in the **Event Wait List** ...

High Priority



Tasks Waiting for Events Are Placed in a Wait List

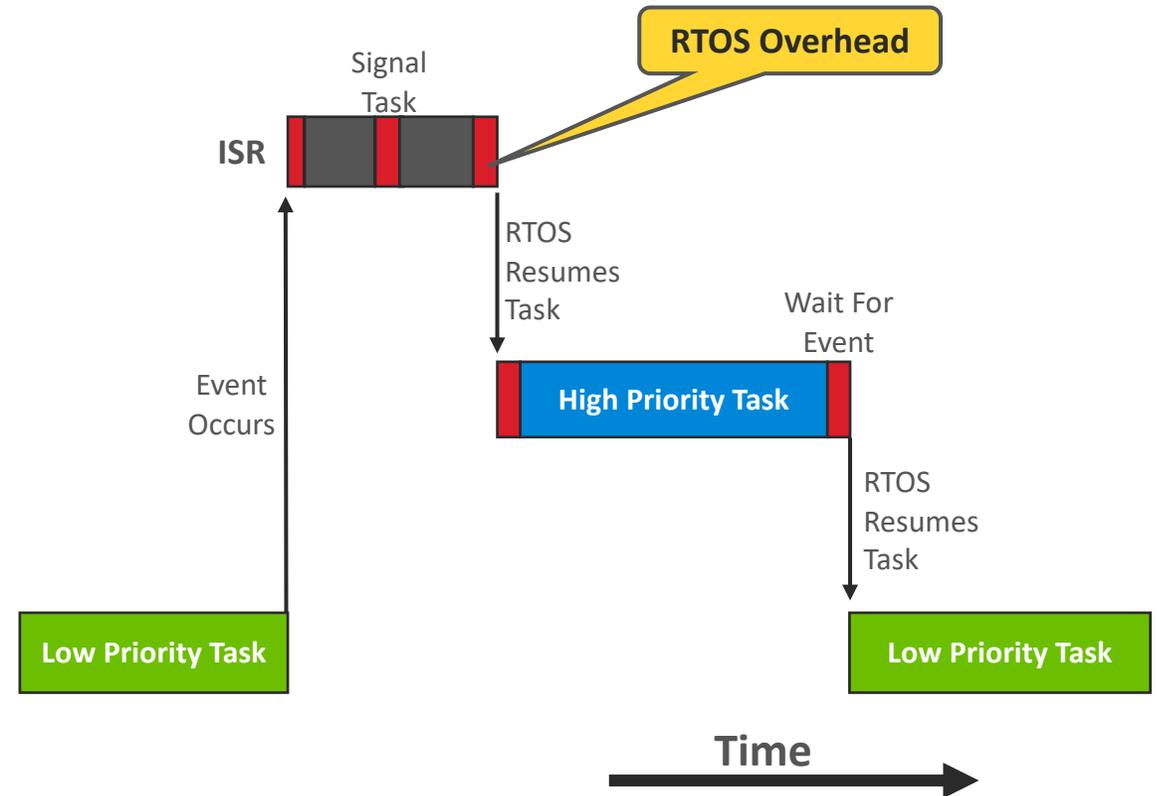


RTOSs Are Preemptive

```
void Low_Prio_Task (void)
{
  Task initialization;
  while (1) {
    Setup to wait for event;
    Wait for event to occur;
    Perform task operation;
  }
}
```

```
void ISR (void)
{
  Entering ISR;
  Perform Work;
  Signal or Send Message to Task;
  Perform Work; // Optional
  Leaving ISR;
}
```

```
void High_Prio_Task (void)
{
  Task initialization;
  while (1) {
    Setup to wait for event;
    Wait for event to occur;
    Perform task operation;
  }
}
```

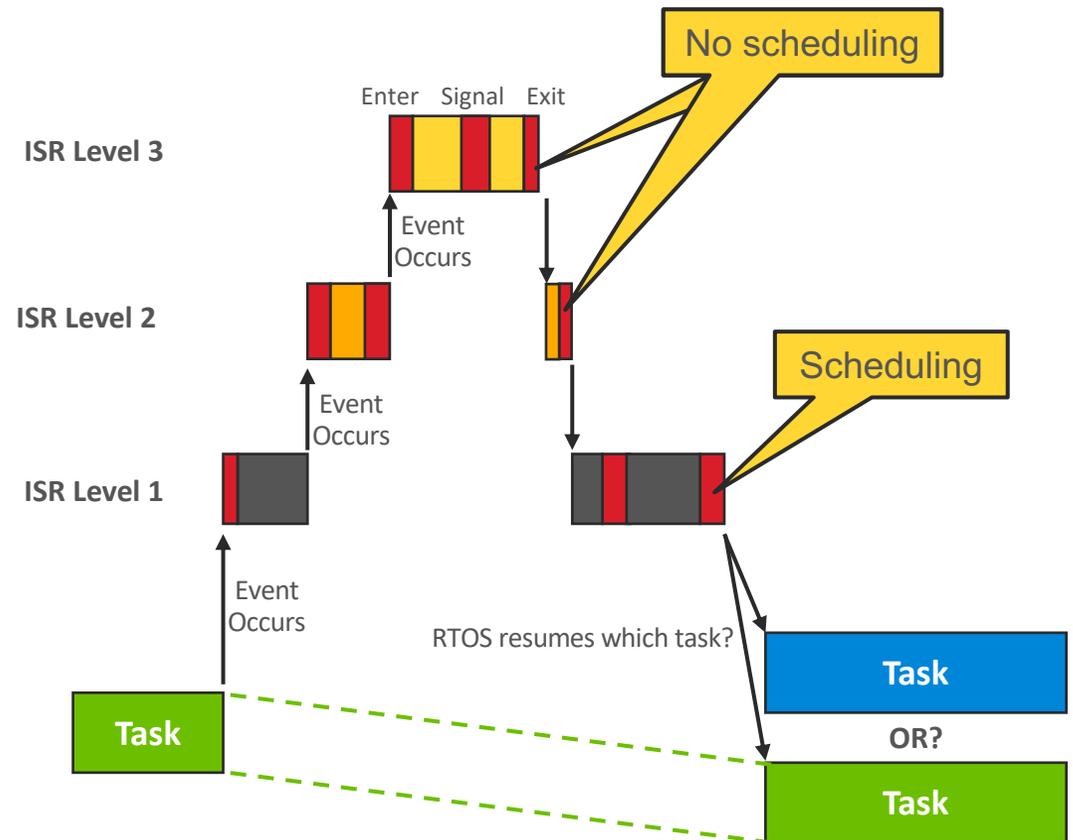


Interrupts Must Notify the RTOS

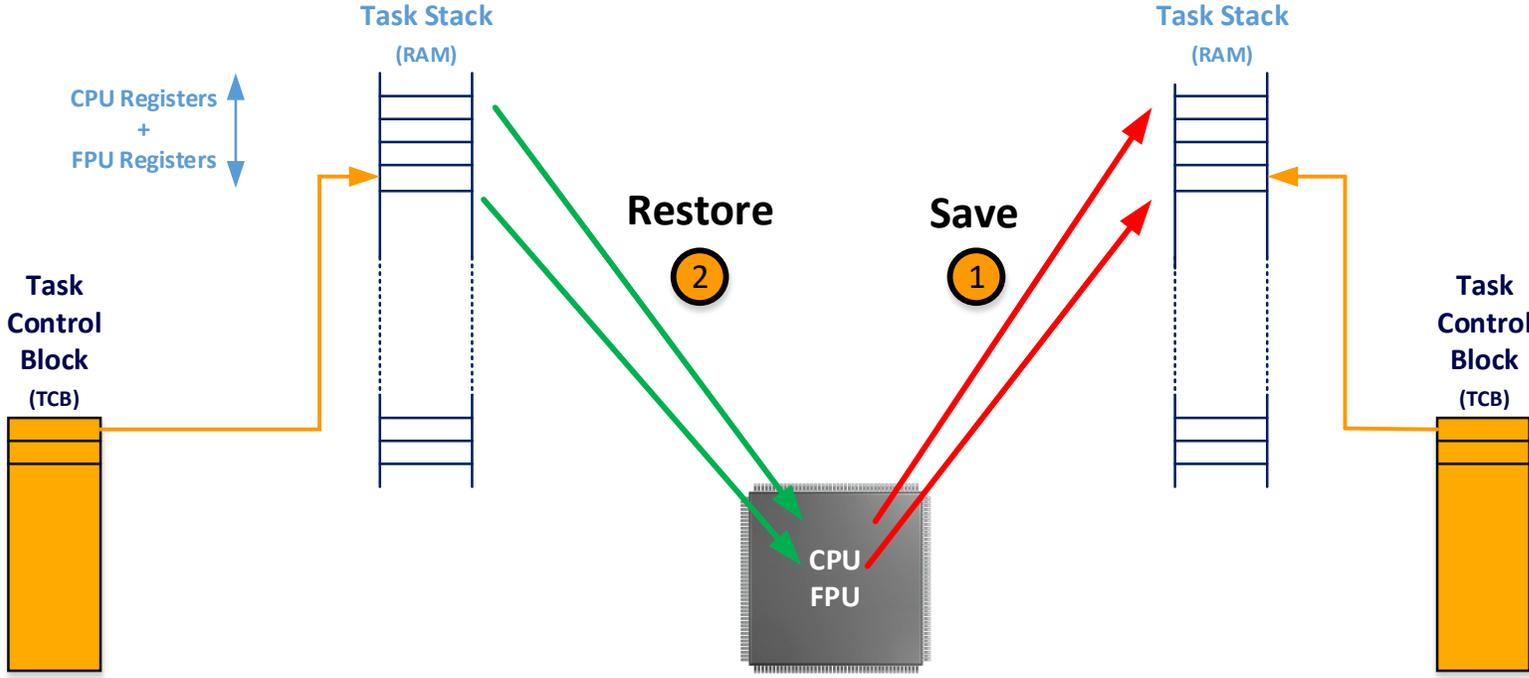
- Oftentimes, interrupts are **events** that tasks are wait for
- Interrupts are **more important** than tasks
 - Assuming, of course, that interrupts are enabled
- **Kernel Aware (KA) ISRs:**
 - Need to notify the RTOS of ISR **entry** and **exit**
 - Allows for nesting ISRs and avoid multiple scheduling

```
void MyISR (void)
{
    Entering ISR;
    :
    Signal or send a message to a MyTask;
    :
    Leaving ISR;
}
```

- ISRs can be written directly in C on Cortex-M CPUs



A Context Switch Occurs When the RTOS Decides to Run Another Task

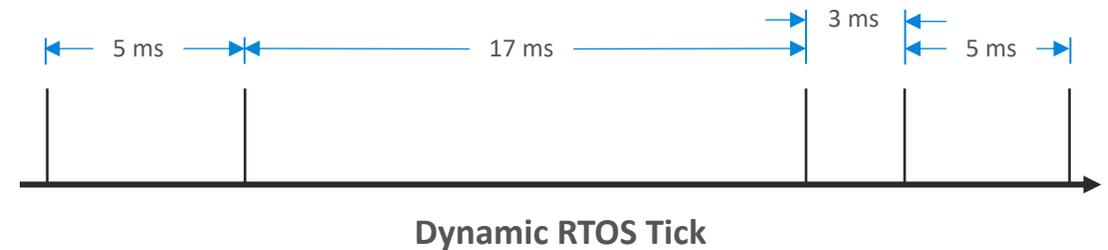
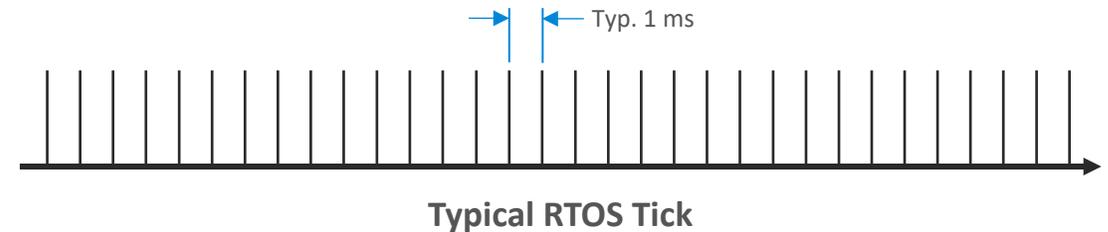


Context Switch

Example using Cortex-M4

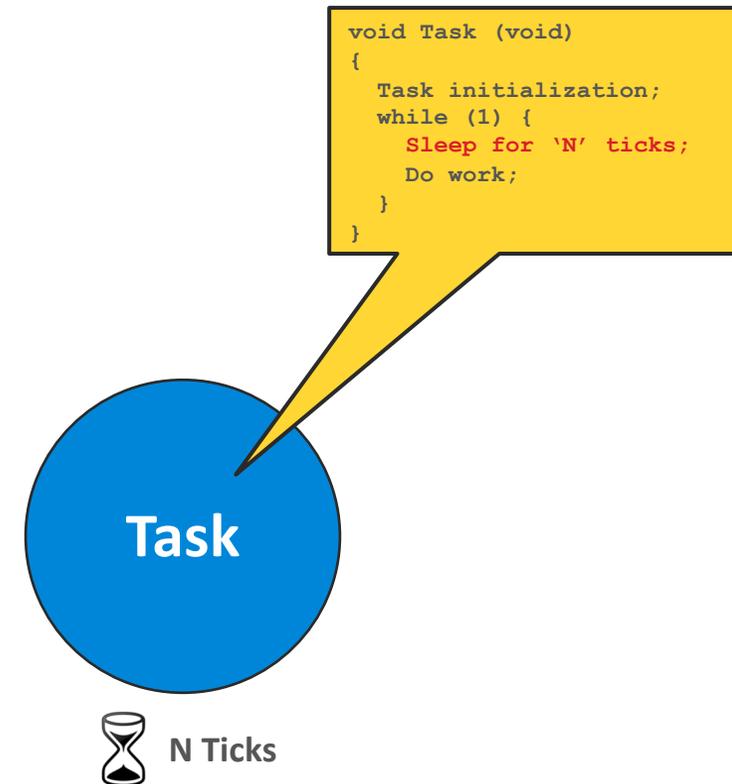
Most RTOSs Have a Periodic Time Source

- Most RTOS have a time-based interrupt
 - Called the **System Tick** or **Clock Tick**
 - Requires a hardware timer
- The System Tick is used to provide **coarse**:
 - Delay (or sleep)
 - Timeouts on **Wait for Event** RTOS APIs
- A System Tick is **not** mandatory!
 - If you don't need time delays or timeouts you can remove it
- Typically interrupts at regular intervals
 - Not power-efficient
 - Dynamic tick (a.k.a. tick suppression) is more efficient
 - Requires reconfiguring the tick timer at each interrupt



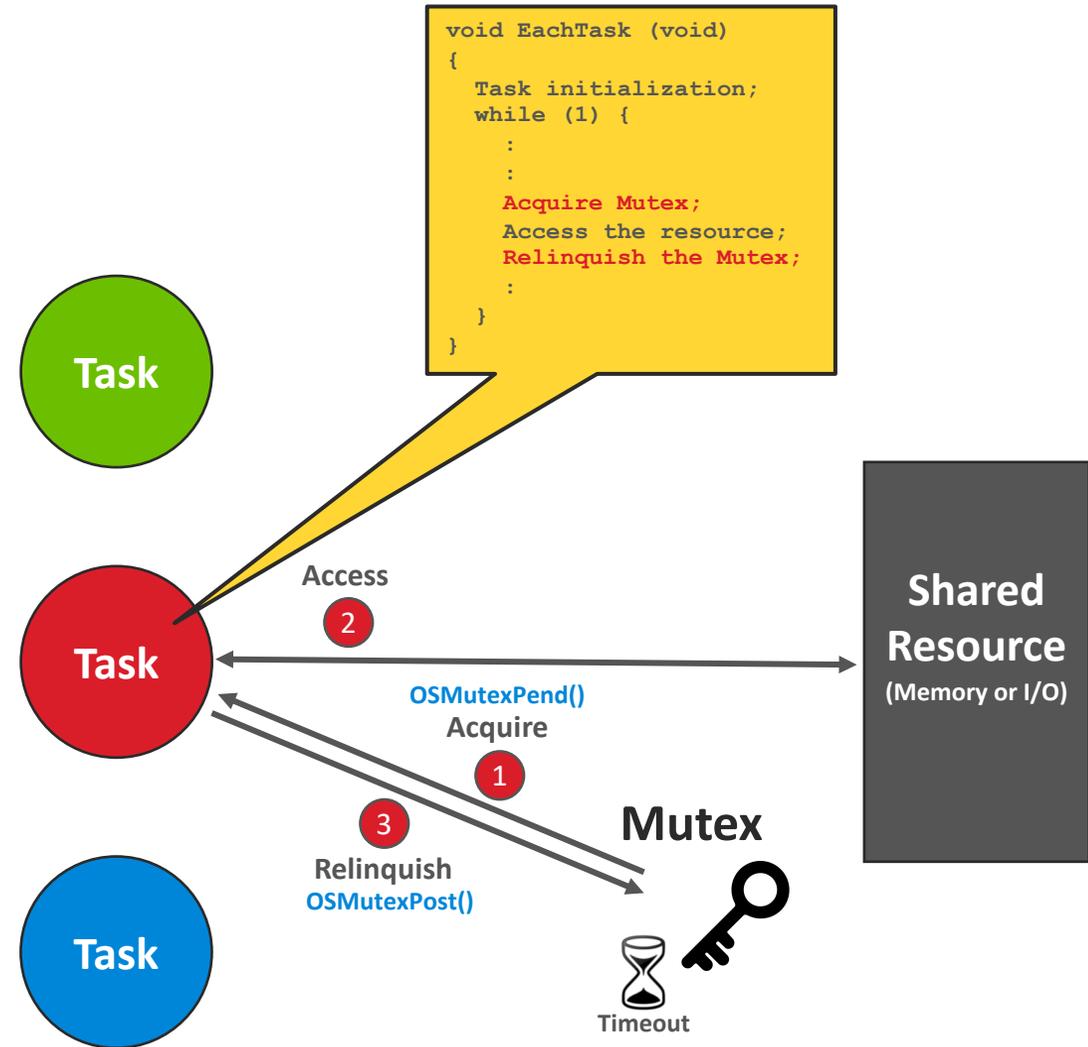
RTOS Services – Time Delays (i.e. Sleep)

- A task can put itself to **sleep** by calling RTOS APIs:
 - `OSTimeDly()` // Delay for N ticks
 - `OSTimeDlyHMSM()` // Delay for Hours, Minutes, Seconds, Milliseconds
- Can be used to **wake up** a task at regular intervals
 - Control loops
 - Updating a display
 - Scanning a keyboard
 - Letting other tasks a chance to run
 - Etc.



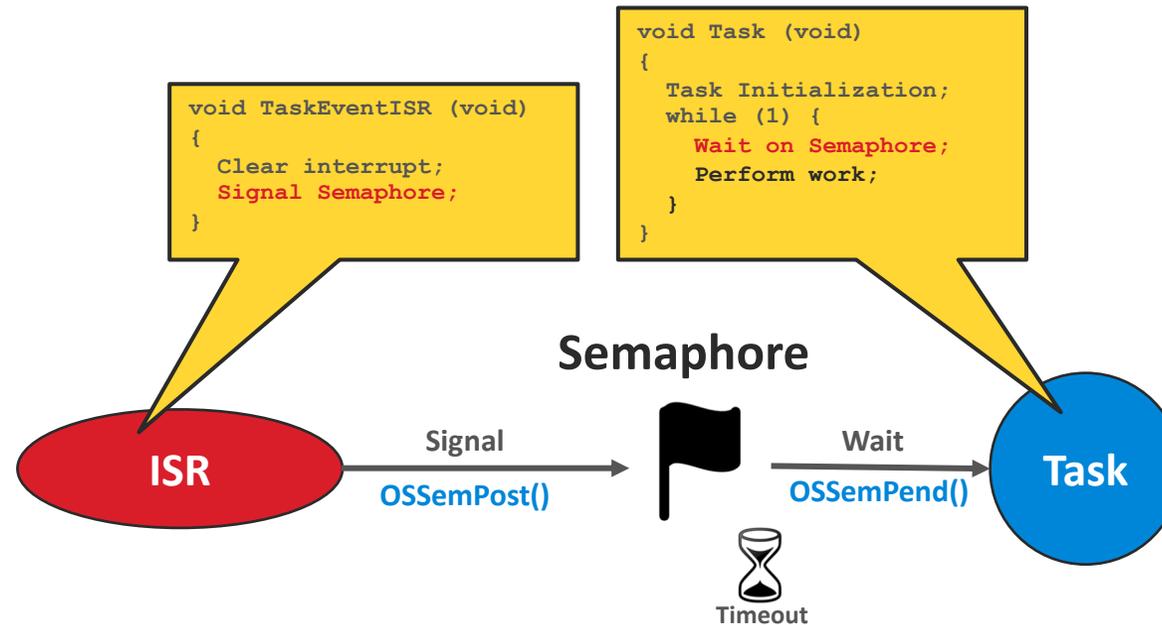
RTOS Services – Sharing a Resource Using a Mutex

- What is a resource?
 - Shared memory, variables, arrays, structures
 - I/O devices
- RTOSs typically provide resource sharing APIs
 - Mutex have built-in **priority inheritance**
 - Eliminates unbounded priority inversions
 - There could be multiple mutexes in a system
 - Each protecting access to a different resource



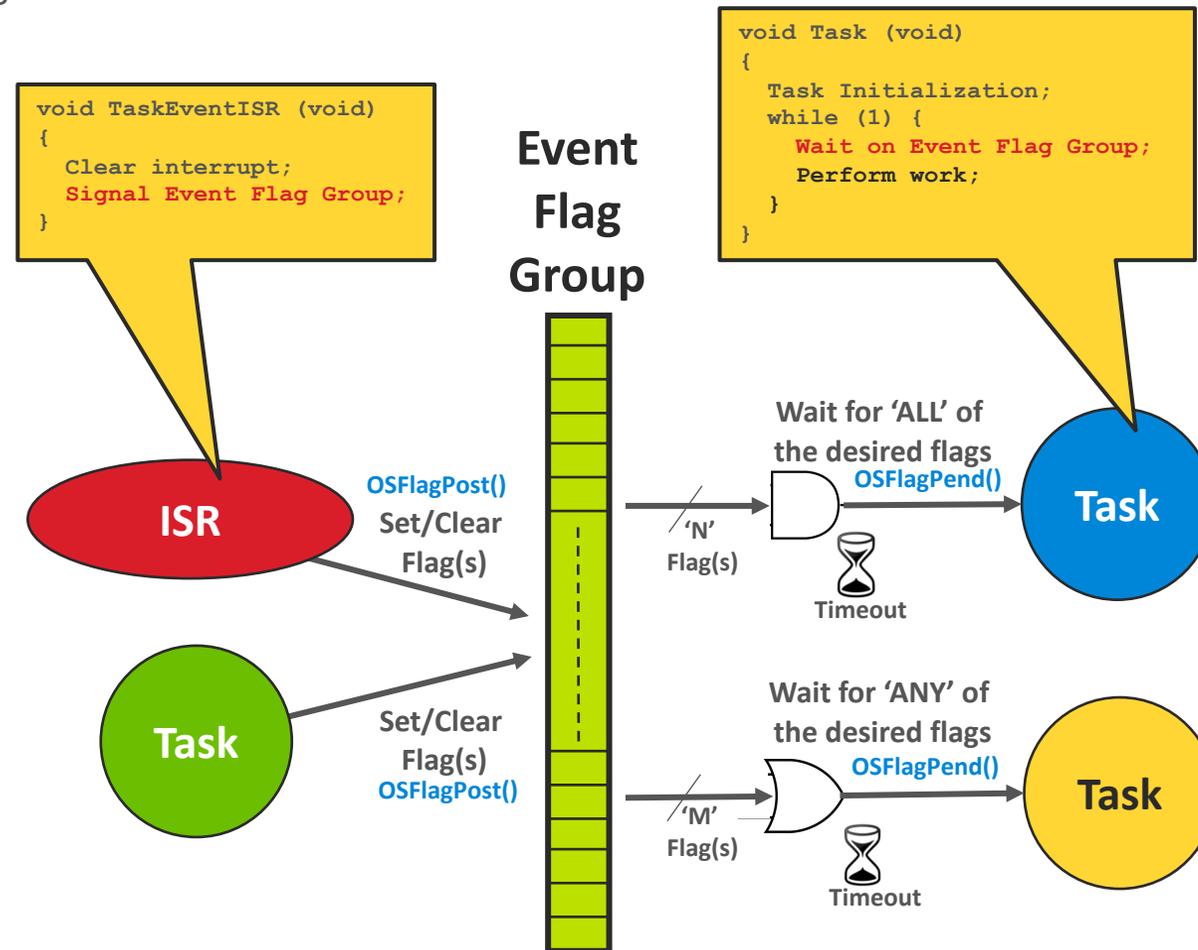
RTOS Services – Signaling a Task Using a Semaphore

- Semaphores can be used to **signal** a task
 - Called from ISR or Task
 - Does not contain data



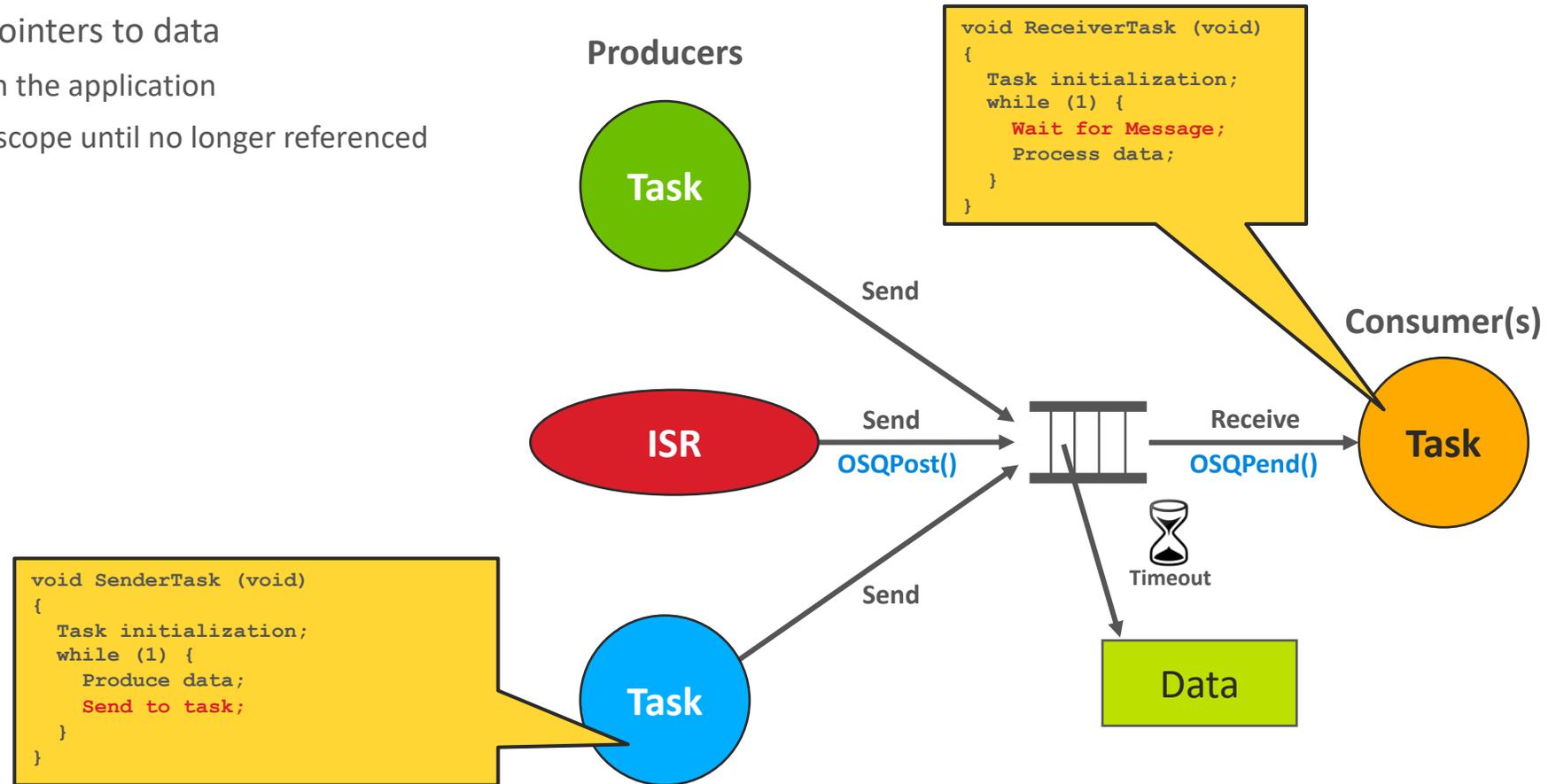
RTOS Services – Signaling Task(s) Using Event Flags

- **Event Flags** are a grouping of bits used to signal the occurrence of more than one events
 - Signals from ISRs or Tasks
 - Only tasks can wait for events

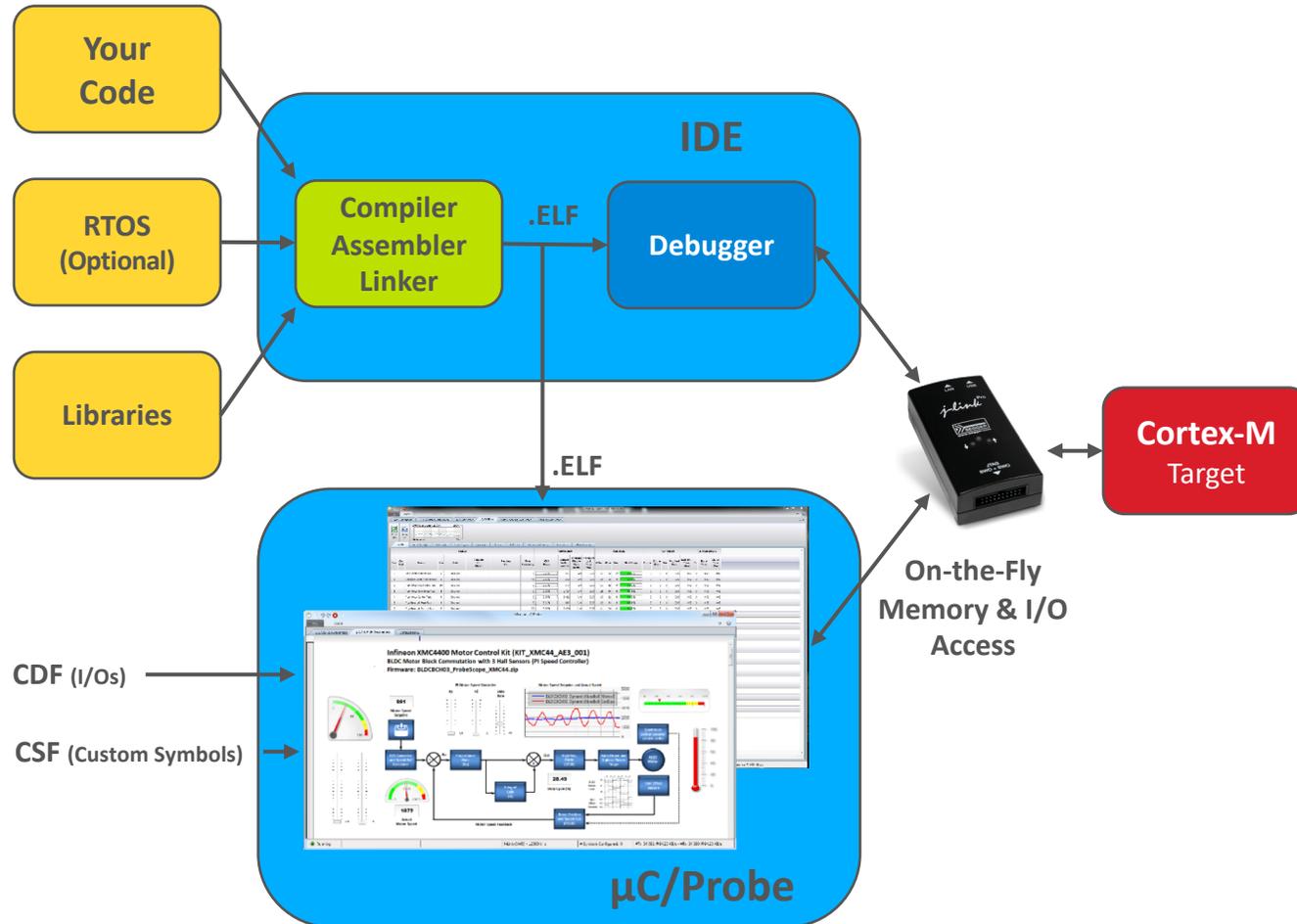


RTOS Services – Sending Messages to Task(s)

- Messages can be sent from an ISR or a task to other task(s)
- Messages are typically pointers to data
 - The data sent depends on the application
 - The data must remain in scope until no longer referenced



Debugging RTOS-Based Systems – μ C/Probe, Graphical Live Watch[®]



- Seeing inside an embedded system
 - Non-intrusive
- Display or change **ANY** values numerically or graphically
- A universal **tool** that interfaces to any target:
 - 8-, 16-, 32-, 64-bit and DSPs
 - No CPU intervention with Cortex-M
 - Requires target resident code if not using the debug port:
 - RS232C
 - TCP/IP
 - USB
- For **bare metal** or **RTOS-Based** applications
 - Micrium's RTOS and TCP/IP awareness

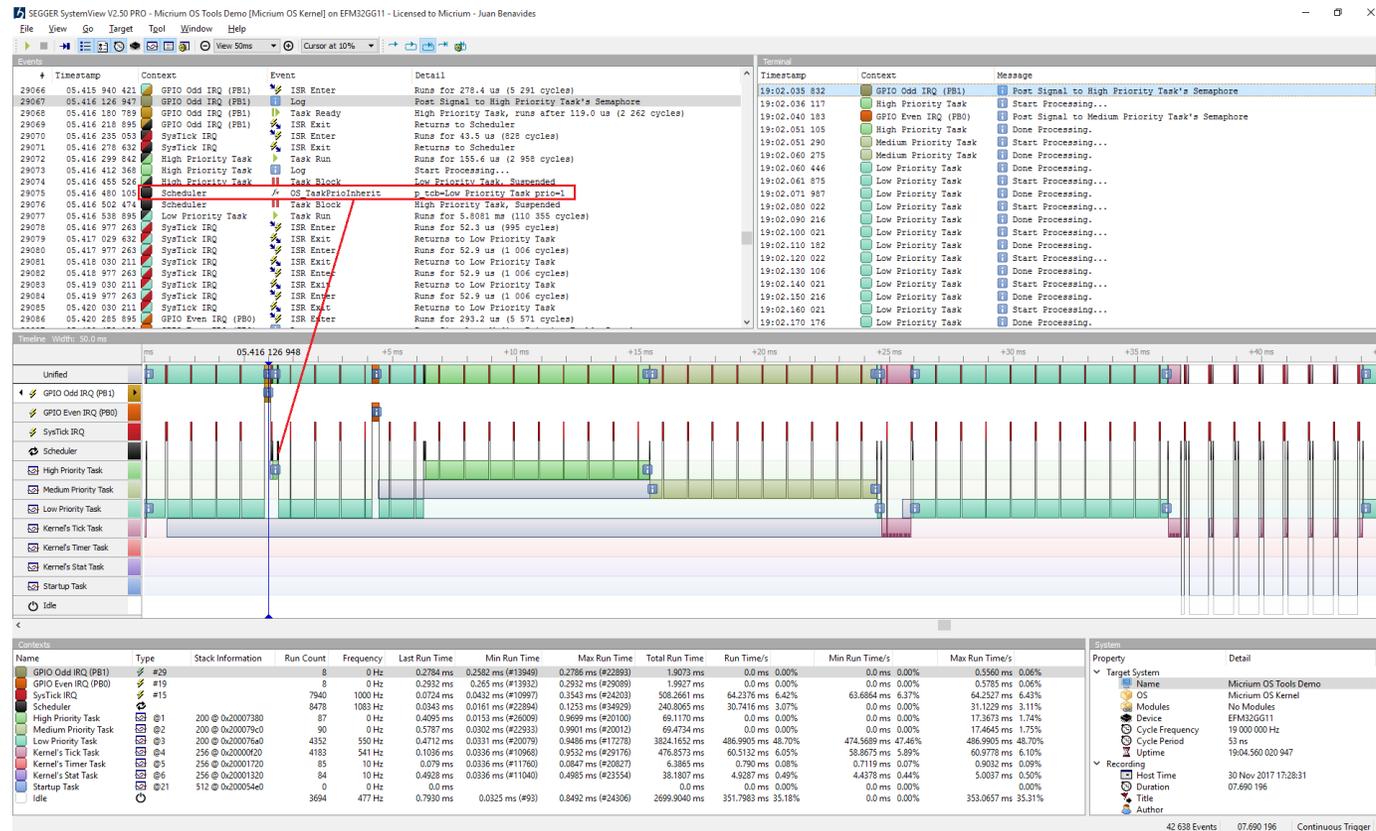
Debugging RTOS-Based Systems – μ C/Probe, Graphical Live Watch[®]

Micrium's μ C/Probe™



- RTOS awareness
 - Task state stack usage
 - CPU usage
 - Run counters
 - Interrupt disable time
 - Kernel objects
 - Etc.
- Built-in 8 channel oscilloscope
- Display live data in Microsoft Excel
- Scripting
- Joystick Interface
- Multimedia support
- Etc.

Debugging RTOS-Based Systems – Segger's SystemView



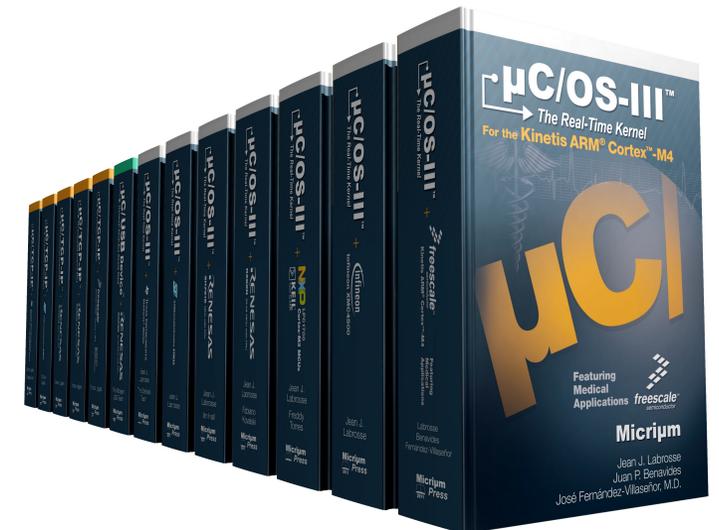
- Displays the execution profile of RTOS-based systems
- Displayed **live**
 - Trigger on any task or ISR
- Visualizing the execution profile of an application
- Helps confirm the expected behavior of your system
- Measures CPU usage on a per-task basis
 - Min/Max/Avg task run time
 - Counts the number of task executions
- Display the occurrence of 'events' in your code
- Traces can be saved for post-analysis or record keeping

Do You Need an RTOS?

- Do you have some **real-time requirements**?
- Do you have **independent tasks**?
 - User interface, control loops, communications, etc.
- Do you have tasks that could **starve other tasks**?
 - e.g. updating a graphics display, receiving an Ethernet frame, encryption, etc.
- Do you have **multiple programmers** working on different portions of your project?
- Is **portability and reuse** important?
- Does your product need **additional middleware** components?
 - TCP/IP stack, USB stack, GUI, File System, Bluetooth, etc.
- Do you have **enough RAM** to support multiple tasks?
 - Flash memory is rarely a concern because most embedded systems have more Flash than RAM
- Are you using a **32-bit CPU**?
 - You should consider using an RTOS

Development Tools and Books

- Silicon Labs Integrated Development Environment (FREE):
 - <https://www.silabs.com/products/development-tools/software/simplicity-studio>
- Silicon Labs Development Boards:
 - <https://www.silabs.com/products/development-tools/mcu>
- Silicon Labs / Micrium OS Kernel (FREE when using Silicon Labs chips):
 - <https://www.silabs.com/products/development-tools/software/micrium-os>
- Micrium's μ C/Probe, Graphical Live Watch[®] (FREE Educational Version):
 - <https://www.micrium.com/ucprobe/trial/>
- Segger's SystemView (FREE Evaluation Version):
 - <https://www.segger.com/downloads/free-utilities/>
- Micrium Books (FREE PDF downloads):
 - <https://www.micrium.com/books/ucosiii/>



Thank you!

SILABS.COM

