An Introduction to Real-Time Operating Systems (a.k.a. RTOSs)

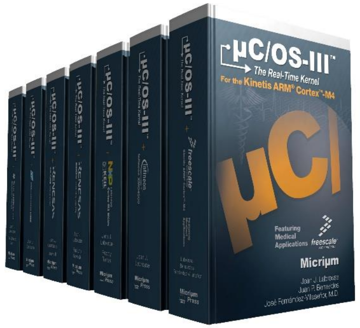
MARCH 2019

JEAN J. LABROSSE

Introduction

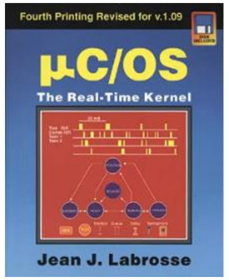
Author

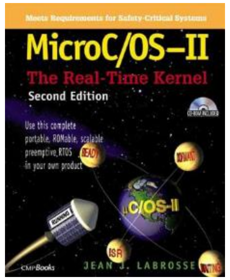
µC/OS series of software and books
Numerous articles and blogs



Lecturer

Conferences
Training





Entrepreneur

Micrium founder (acquired by Silicon Labs in 2016)

SILICON LABS

Embedded Systems Innovator

Embedded Computer Design Innovator of the Year award (2015)

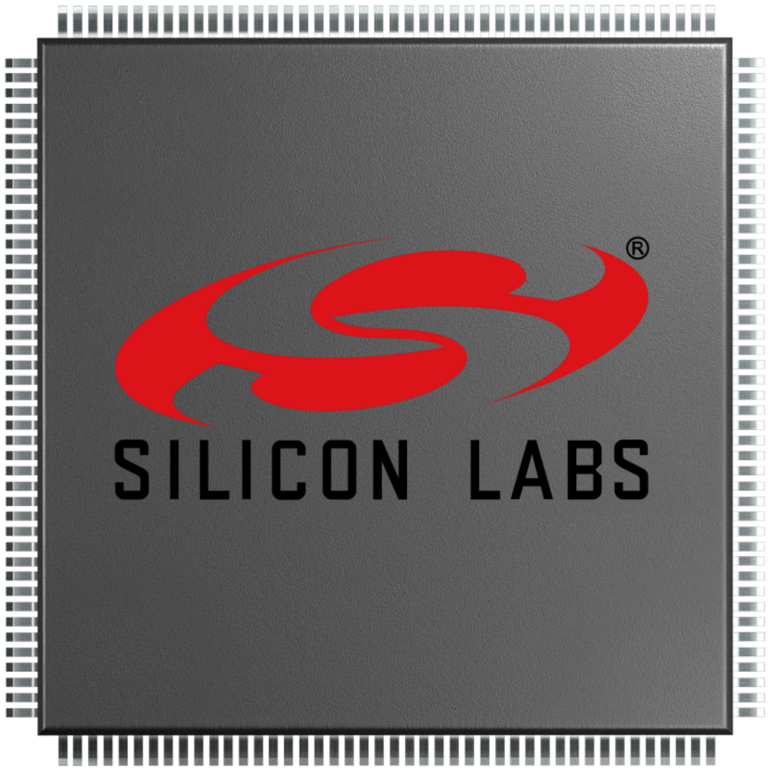
Jean.Labrosse@SiLabs.com

Distinguished Engineer, Software Architect

www.silabs.com www.micrium.com

Assumptions about attendees

- Understand Microprocessors
  - 8-, 16- or 32-bit CPUs
  - Instruction Sets
  - Memory
  - I/Os (Peripherals)
  - Interrupts
- Computer Science
  - Knowledge of C and assembly language
    - Compilers, Assemblers, Linkers
  - Understand Data Structure
  - Familiar with Software Debugging



Agenda

Agenda

- About Silicon Labs / Micrium
- Bare Metal Systems
- What is an RTOS?
- RTOS basics
- RTOS Services
- Seeing Inside Live Embedded Systems
- Debugging RTOS-Based Systems
- RTOS Usage Examples
- Recommendations
- References

About **Silicon Labs / Micriμm**

Silicon Labs - A Global Company

The Leader in IoT Wireless Connectivity

Bluetooth

Proprietary

Thread

Wi-Fi

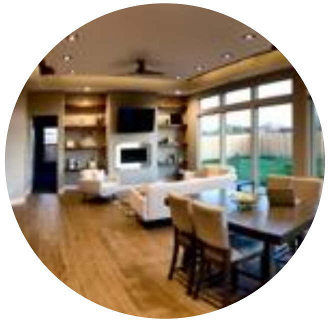
Zigbee

Z-Wave

Serving a Broad Range of Customers and Application Areas

30 million hours saved
yearly with smart metering applications

We've shipped
more than 150 million
mesh networking devices



Boosted energy capacity by 36 GW in 5 years in
7.3 million solar inverters

We help coordinate
90% of Internet traffic

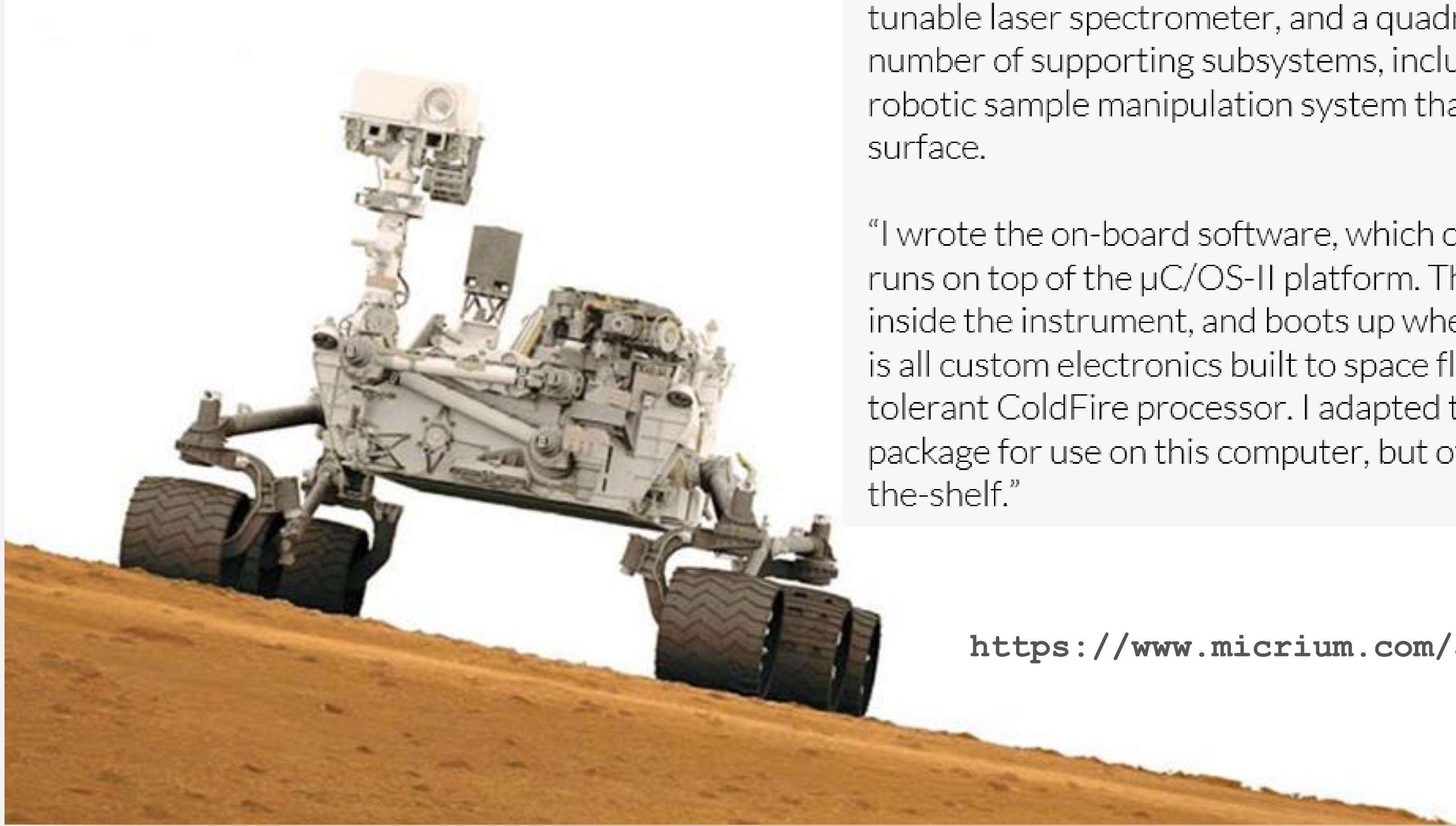
We're in more than
360,000 EV/HEV cars

On board 100% of cherry red electric
Tesla roadsters currently
orbiting the sun

Introducing **Micriμm**

- Provider of High Quality Embedded Software
  - RTOS, protocol stacks and other components
  - Remarkably clean code
  - Outstanding documentation
  - Top-notch technical support
  - Debug tools
- Founded in 1999, Acquired by Silicon Labs in 2016.
- Based in the US (South Florida)
- Provider of high-quality embedded software
- **FREE** for **Educational Use**
  - Licensed for commercial use

µC/OS-II – On Mars



Tom Nolan, Operations Engineer
NASA Jet Propulsion Laboratory

"Sample Analysis at Mars is a suite of three instruments: a gas chromatograph, a tunable laser spectrometer, and a quadrupole mass spectrometer, together with a number of supporting subsystems, including vacuum pumps, pyrolysis ovens, and a robotic sample manipulation system that handles solid samples from the planetary surface.

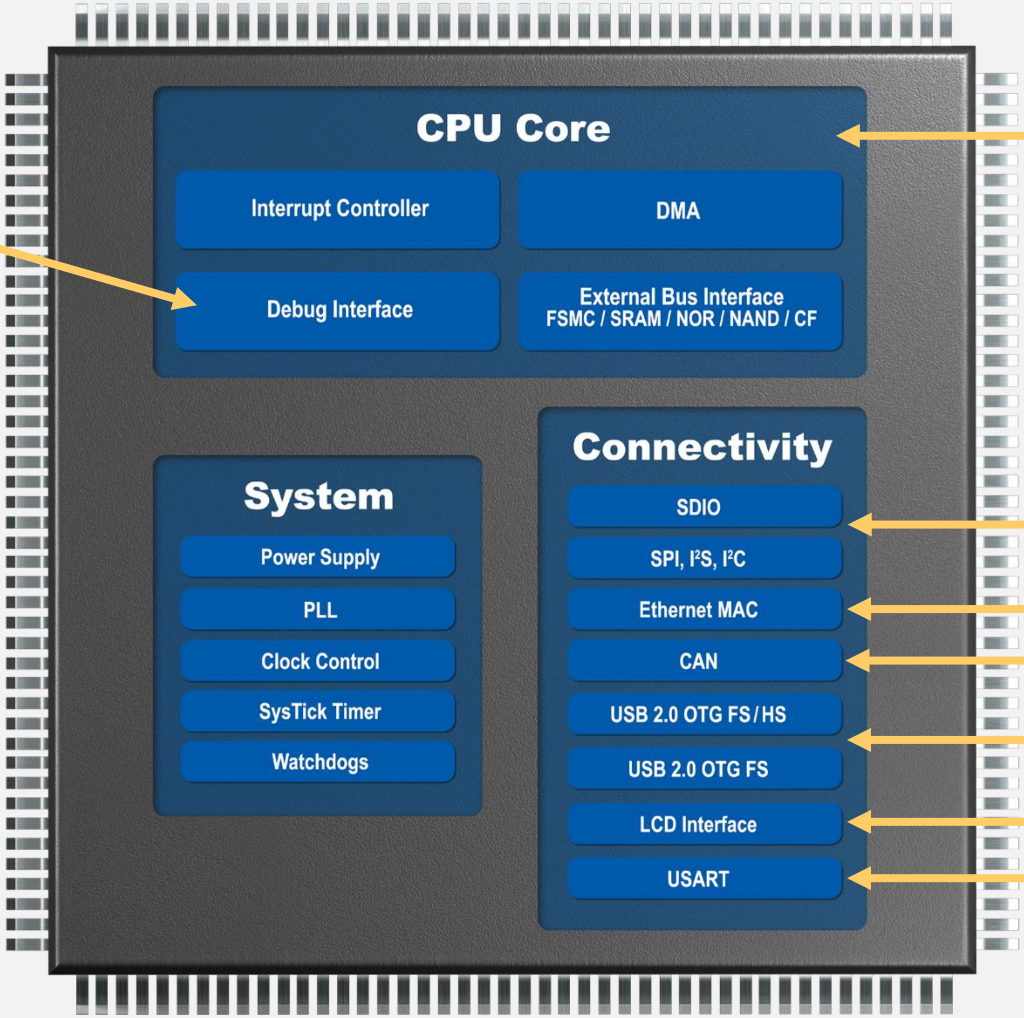
"I wrote the on-board software, which consists of about 20,000 lines of C code, and runs on top of the µC/OS-II platform. The software resides in nonvolatile memory inside the instrument, and boots up when power is applied. The on-board computer is all custom electronics built to space flight standards, and the CPU is a radiation-tolerant ColdFire processor. I adapted the Micrium ColdFire board-support package for use on this computer, but other than that, the operating system is off-the-shelf."

https://www.micrium.com/about/customer-stories/curiosity/

Micriµm – Embedded Software

Tools

µC/Probe



Software Components

Kernel (RTOS)

File System

TCP/IP

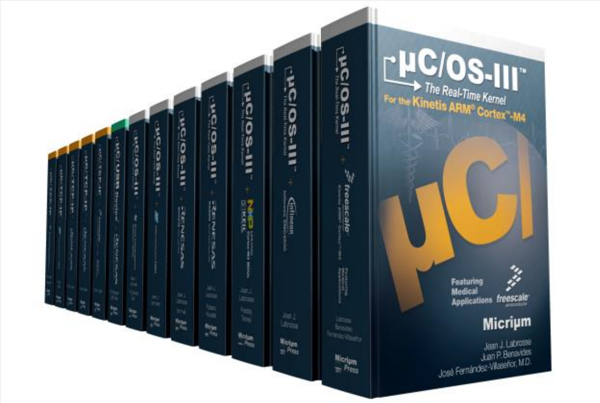
CAN

USB (Host and Device)

GUI

Modbus

Books



Micriμm - A Tradition of Quality

1992 – Original μC/OS released

1993 – μC/OS first RTOS used with ARM processor

2000 – μC/OS-II first used in DO-178B aerospace project

2009 – MicriumPress publishes initial book in μC/OS-III series

2015 – Top commercial RTOS ranking for Micrium in annual UBM survey

Micriμm – Semiconductor Partners

- ALTERA
- ARM
- ANALOG DEVICES
- Atmel
- CYPRESS Embedded in Tomorrow
- FUJITSU
- infineon
- Microsemi
- NXP
- RENESAS
- SAMSUNG
- SILICON LABS
- ST life.augmented
- TEXAS INSTRUMENTS
- TOSHIBA Leading Innovation
- XILINX

Bare Metal Systems (a.k.a. *Super Loops* or, *Single Threaded*)

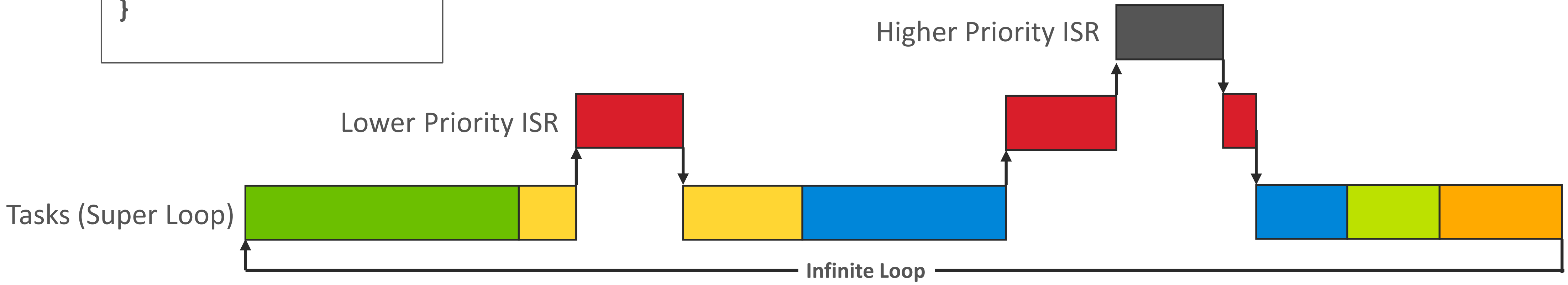
Bare Metal – Super Loop

Low Priority - **High Priority**

```
void  main (void)
{
     Init();
     for (;;) {
          Task_1();
          Task_2();
          Task_3();
          Task_4();
          Task_5();
     }
}
```

```
void  LP_ISR (void)
{
     Clear Interrupt;
     Perform Work;
}
```

```
void  HP_ISR (void)
{
     Clear Interrupt;
     Perform Work;
}
```



Bare Metal - Benefits

- Used in fairly simple applications
- You only need a single stack
  - Set the SP once at startup
  - Requires less RAM
- High performance
  - Highly responsive to interrupts
    - But, ISRs often do too much of the work that should be handled by a task
  - Interrupt disable time dictated by your application
- You can use non-reentrant functions

Bare Metal - Drawbacks

- Difficult to ensure that each operation will meet its deadlines
  - All code in the **main()** loop has the same priority
- If one function call takes longer than expected, the responsiveness of the whole system can suffer
  - Excessive polling waste CPU time
  - Hardware failure can lock up the application

```
void  main (void)
{
    Initialization;
    while (1) {
        ADC_Read();
        SPI_Read();
        USB_Packet();
        LCD_Update();
        Audio_Decode();
        File_Write();
    }
}
```

?

```
void  ADC_Read (void)
{
    Initialize ADC;
    while (ADC Converter NOT ready) {
        ;
    }
    Process converted value;
}
```

Unexpected delays and possible lockup

Bare Metal - Drawbacks

- High priority code must be placed in ISRs
  - Long ISRs may affect the responsiveness of the system
  - Coordination between ISR and **main()** is difficult

```
void  main (void)
{
    Initialization;
    while (1) {
        ADC_Read();
        SPI_Read();
        USB_Process_Packet();
        LCD_Update();
        Audio_Decode();
        File_Write();
    }
}
```

Could take a long time before the packet gets processed

```
void  USB_ISR (void)
{
    Clear Interrupt;
    Read Packet;
    Indicate packet received;
}
```

Bare Metal - Drawbacks

- The responsiveness of the application can change as you add code
  - Code is often duplicated to compensate for lack of responsiveness
  - Counters are used to limit the execution rate
- Large applications are difficult to maintain
  - Difficult to coordinate the effort of multiple developers and ensure timing requirements are met
  - Changes to one portion of the code can impact another
- Difficult to use protocol stacks
  - Many of the protocol stacks assume an RTOS
- Difficult to do battery management

Code duplication

```
while (1) {
    ADC_Read();
    LCD_Update();
    SPI_Read();
    USB_Packet();
    LCD_Update();
    Audio_Decode();
    File_Write();
    LCD_Update();
}
```

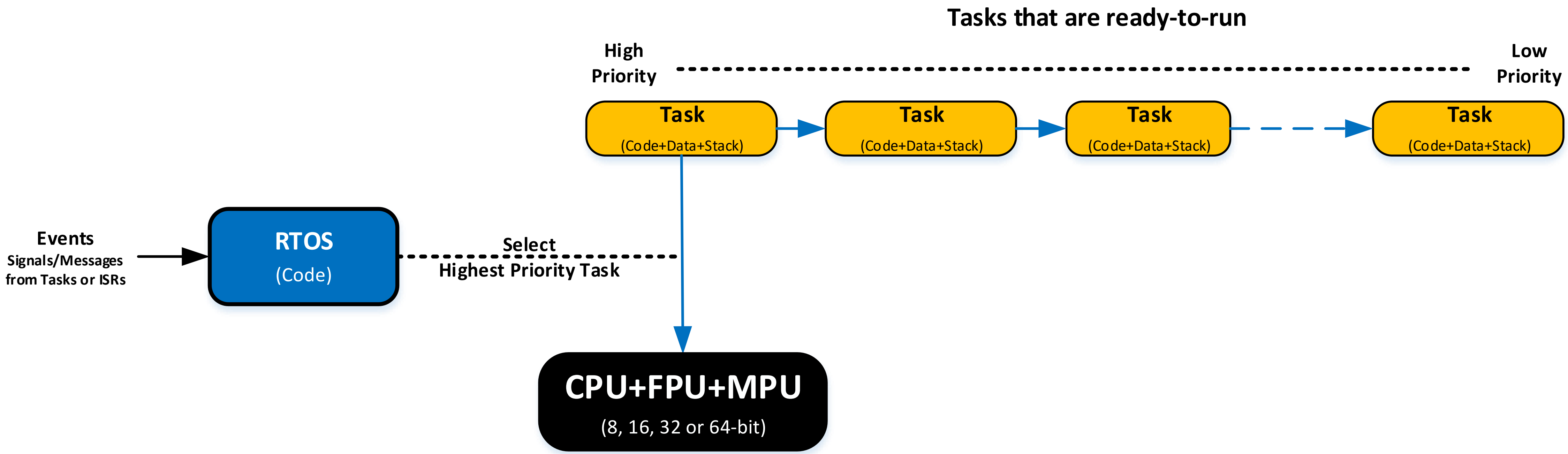
Counters to limit execution rate

```
while (1) {
    ADC_Read();
    if ((i % 64) == 0) {
        SPI_Read();
    }
    USB_Packet();
    LCD_Update();
    if ((i % 32) == 0) {
        Audio_Decode();
    }
    File_Write();
    i++;
}
```

What Is An RTOS? (a.k.a. Real-Time Kernel)

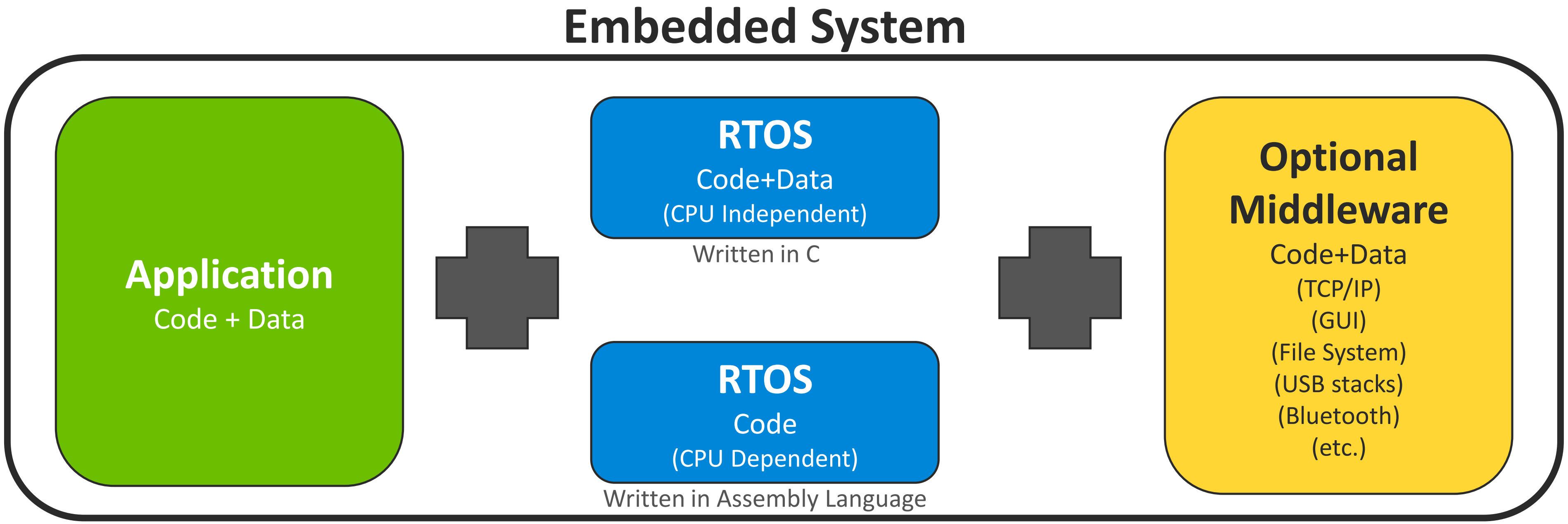
What Is An RTOS? - Multitasking

- 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

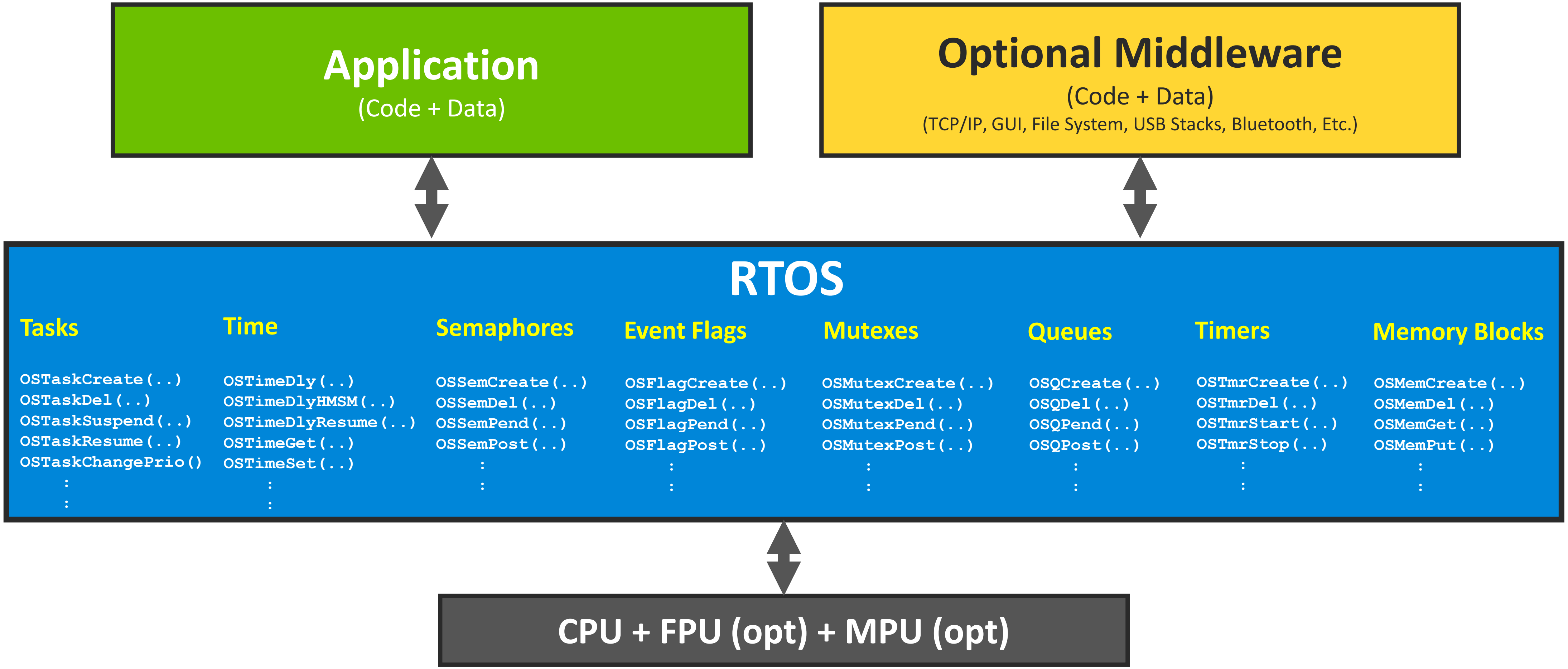


What Is An RTOS? – Code That You Add To Your Application

- An RTOS is either provided in source form or as a library that you link to your code
  - Most RTOSs are written in C
  - Assembly language code is needed to adapt the RTOS to different CPU architectures (called a ***Port***)
    - This is provided by the RTOS supplier

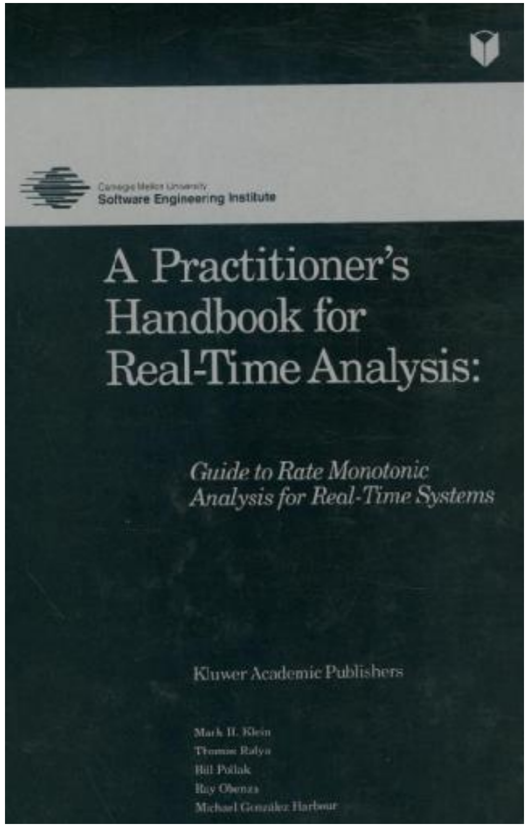


What Is An RTOS? – Provide Services To Your Application



What Is An RTOS? - Benefits

- Creates a **framework** for developing applications
  - Facilitate teams of multiple developers
- 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
  - A task **doesn't consume** any CPU time while waiting – avoids polling
- **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

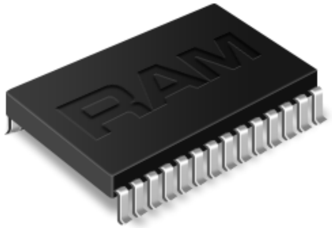


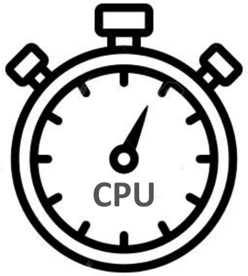
What Is An RTOS? - Benefits

- Provides **services** to your application
  - ISR management
  - Task management
  - Time management
  - Resource management
  - ISR and inter-task communication
  - Memory management
  - Etc.
- RTOSs make it **easy to add middleware** components
  - TCP/IP stack
  - USB stacks
  - File System
  - Graphical User Interface (GUI)
  - Etc.

What Is An RTOS? - Drawbacks

- The **RTOS itself is code** and thus requires more Flash
  - Typically between 6-20 **Kbytes**
- 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 2-5% of the CPU cycles, could be more
- There is a **learning curve** associated with the RTOS you select





What Is An RTOS? – Do You Need One?

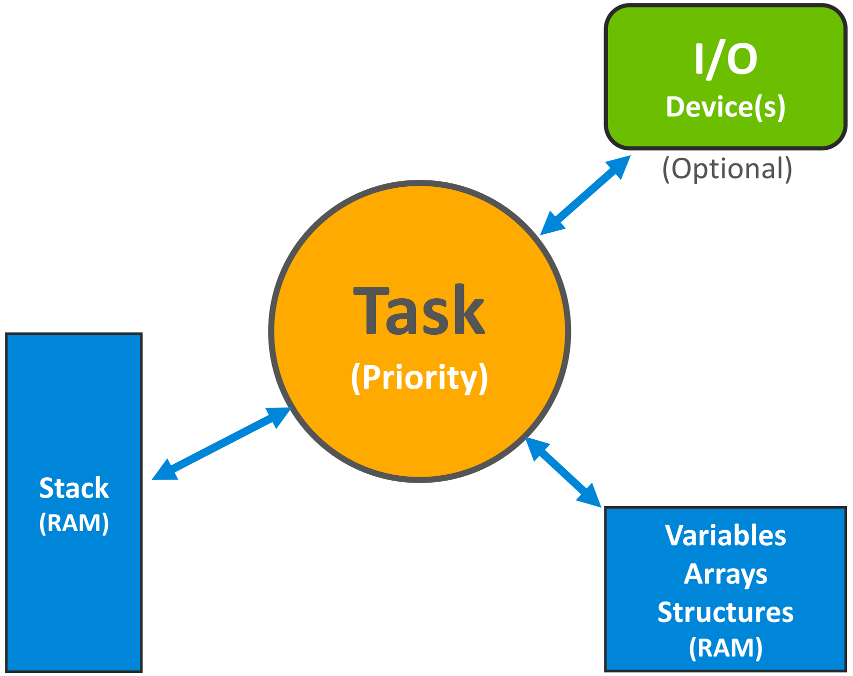
- Do you have **some** real-time requirements?
- Do you have **independent** tasks?
  - User interface, control loops, communications, etc.
- Do you have tasks that **could starve** other tasks?
  - e.g. updating a graphics display, receiving an Ethernet frame, encryption, etc.
- Do you have **multiple 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

RTOS Basics

RTOS Basics – Tasks

- Each task:
  - Is assigned 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
  }
}
```

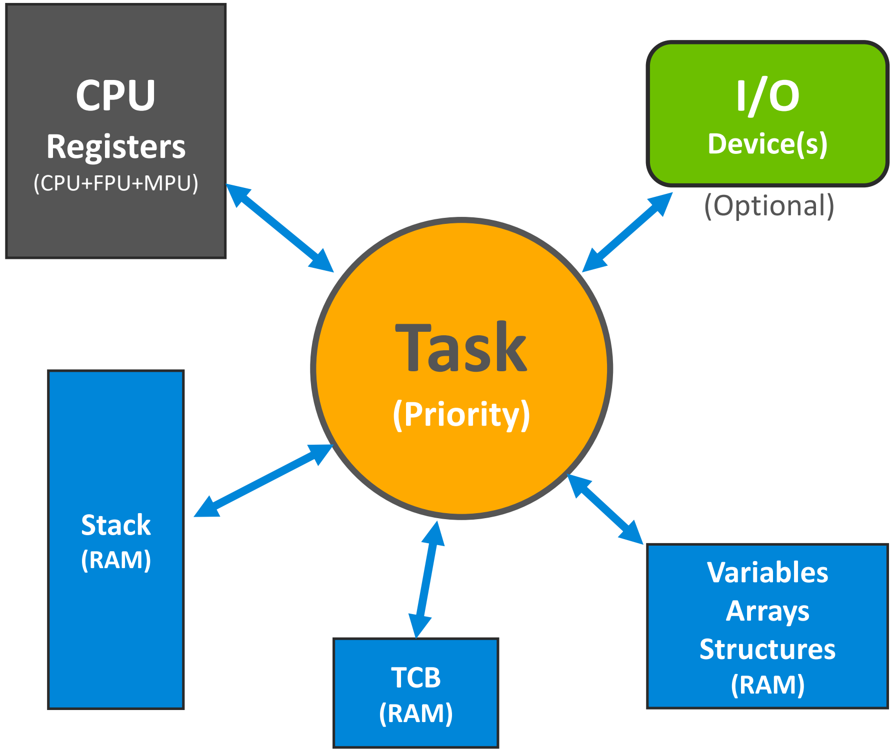


RTOS Basics – Creating A Task

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

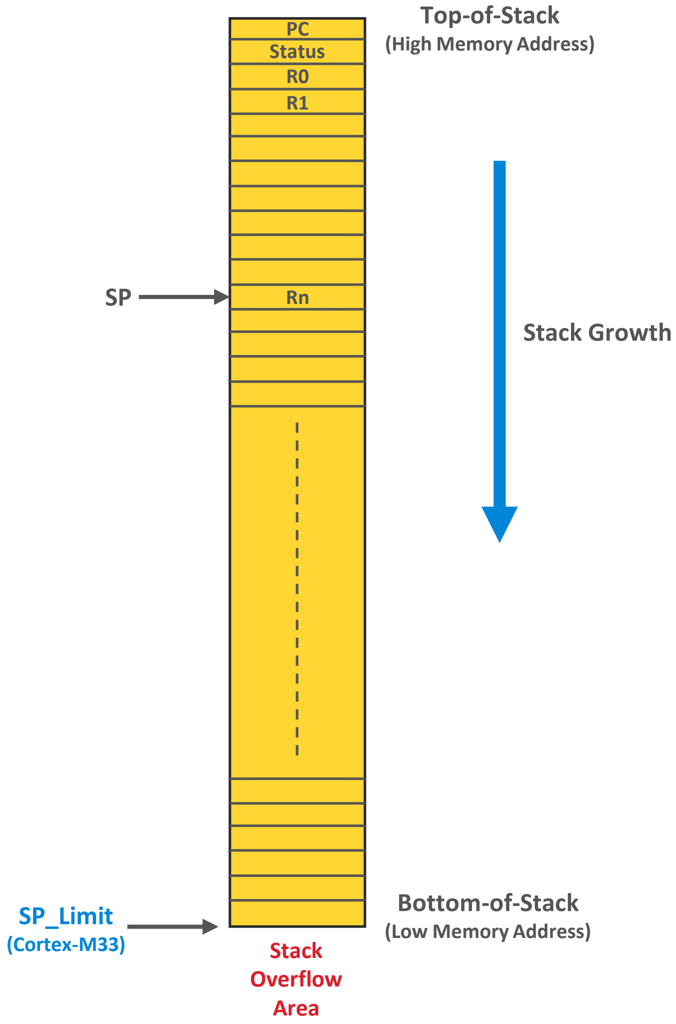
```
void OSTaskCreate (MyTask,            // Address of code
                  &MyTaskStk[0],      // Base of stack
                   MY_TASK_STK_SIZE,// Size of stack
                   MY_TASK_PRIO,      // Task priority
                   :
                   :);
```

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



RTOS Basics – The Task's Stack

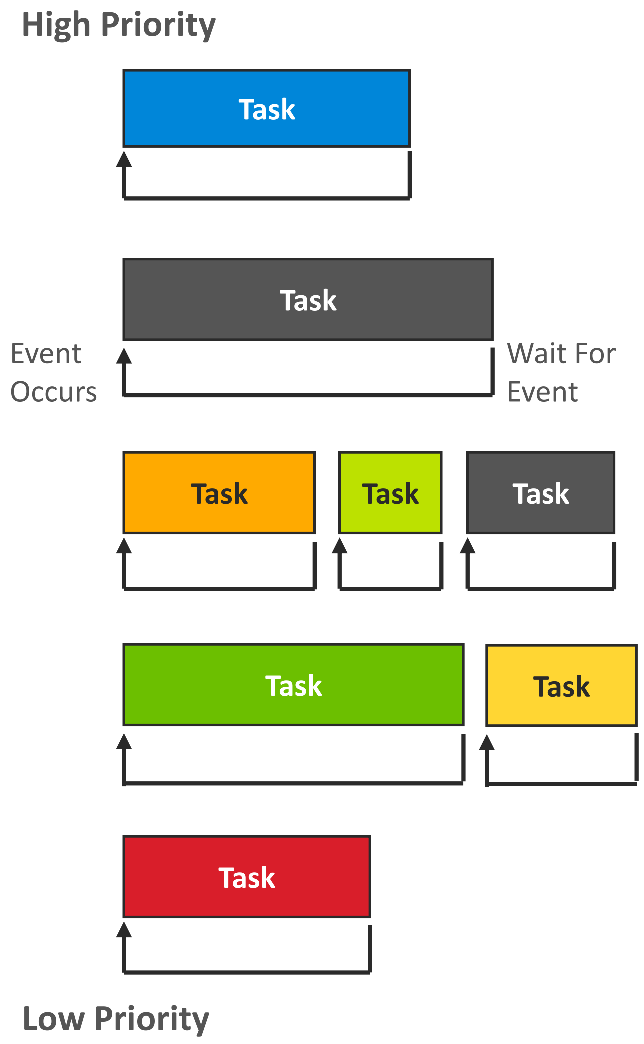
- Each task requires **its own stack**
  - Local variables
  - Return addresses
  - The size depends on what the task does
    - Each task can have a different stack size
- When a task is created:
  - The ***Top-Of-Stack*** is populated by with the initial values of CPU registers
    - R0-Rn, Status Register, PC
    - FPU registers (If the CPU has an FPU)
  - The ***Bottom-of-Stack*** is populated with **canary** values
    - Used to determine stack usage and detect stack overflows
    - An RTOS task can scan each of the task stacks to compute actual CPU usage
- The Cortex-M33 processor has hardware **Stack Limit detection**
  - A fault is generated if the **SP** is changed to be lower than the **SP_Limit**
  - The RTOS can then terminate the offending task



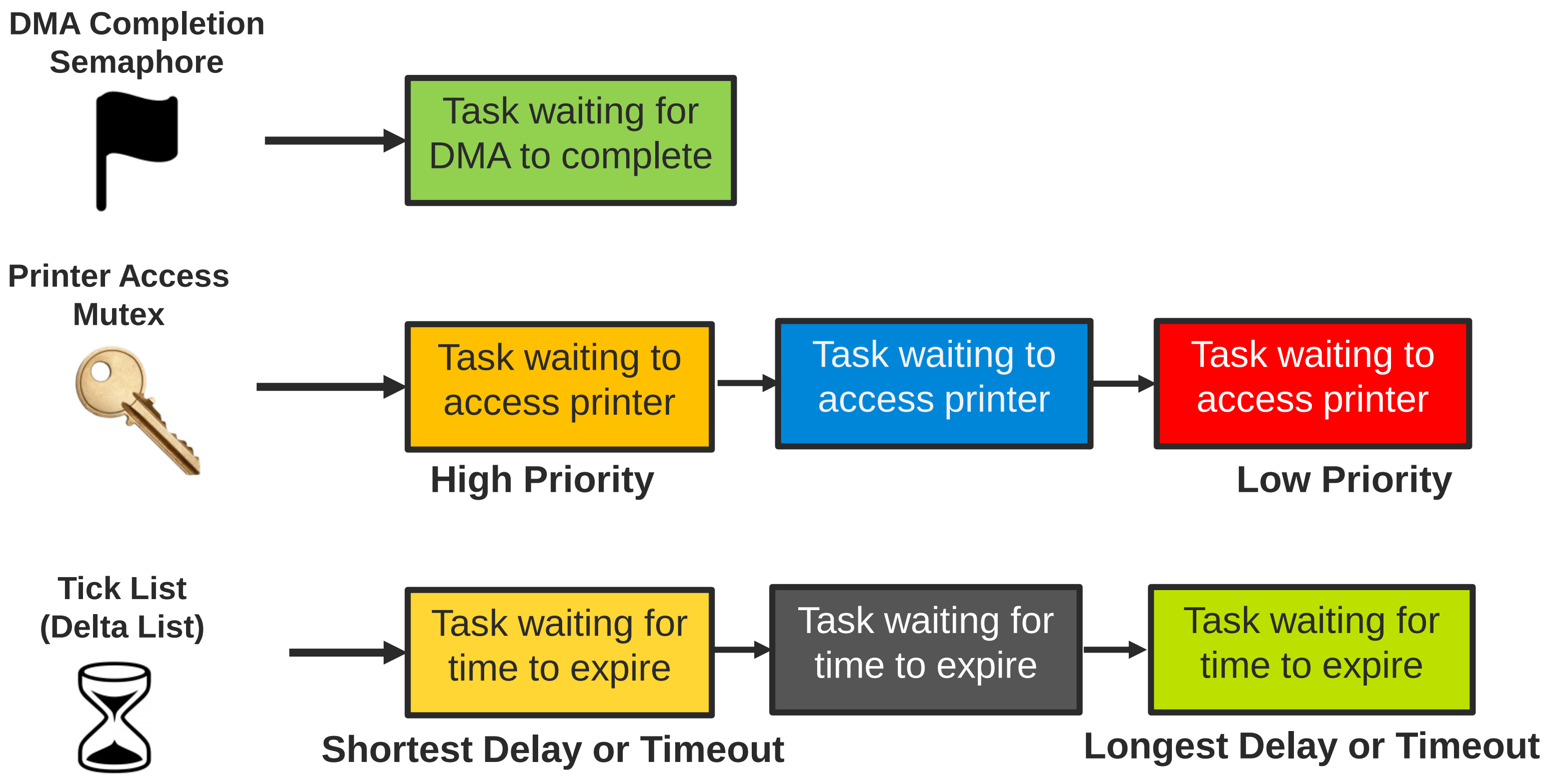
RTOS Basics – Event Driven

```
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*** …



RTOS Basics – Wait Lists



Notes:

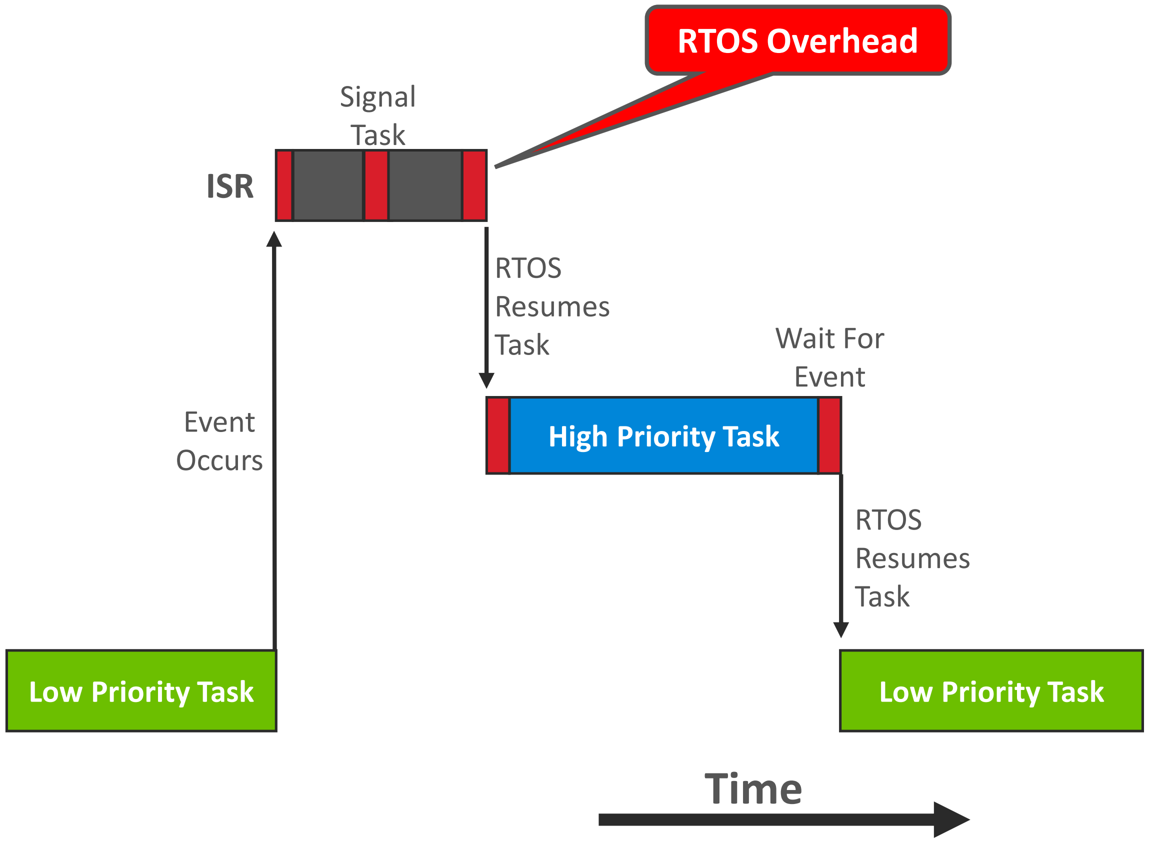
1) List of Task Control Blocks (TCBs)
2) A task can be in 2 lists at the same time (the second one would be the Tick List)

RTOSs are typically 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;
}
```

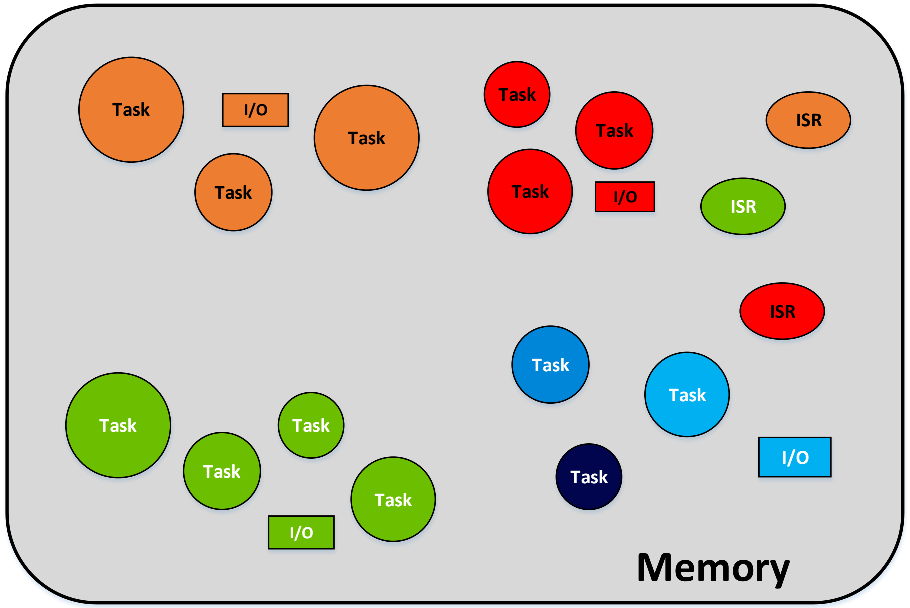
```
void High_Prio_Task (void)
{
  Task initialization;
  while (1) {
    Setup to wait for event;
    Wait for event to occur;
    Perform task operation;
  }
}
```



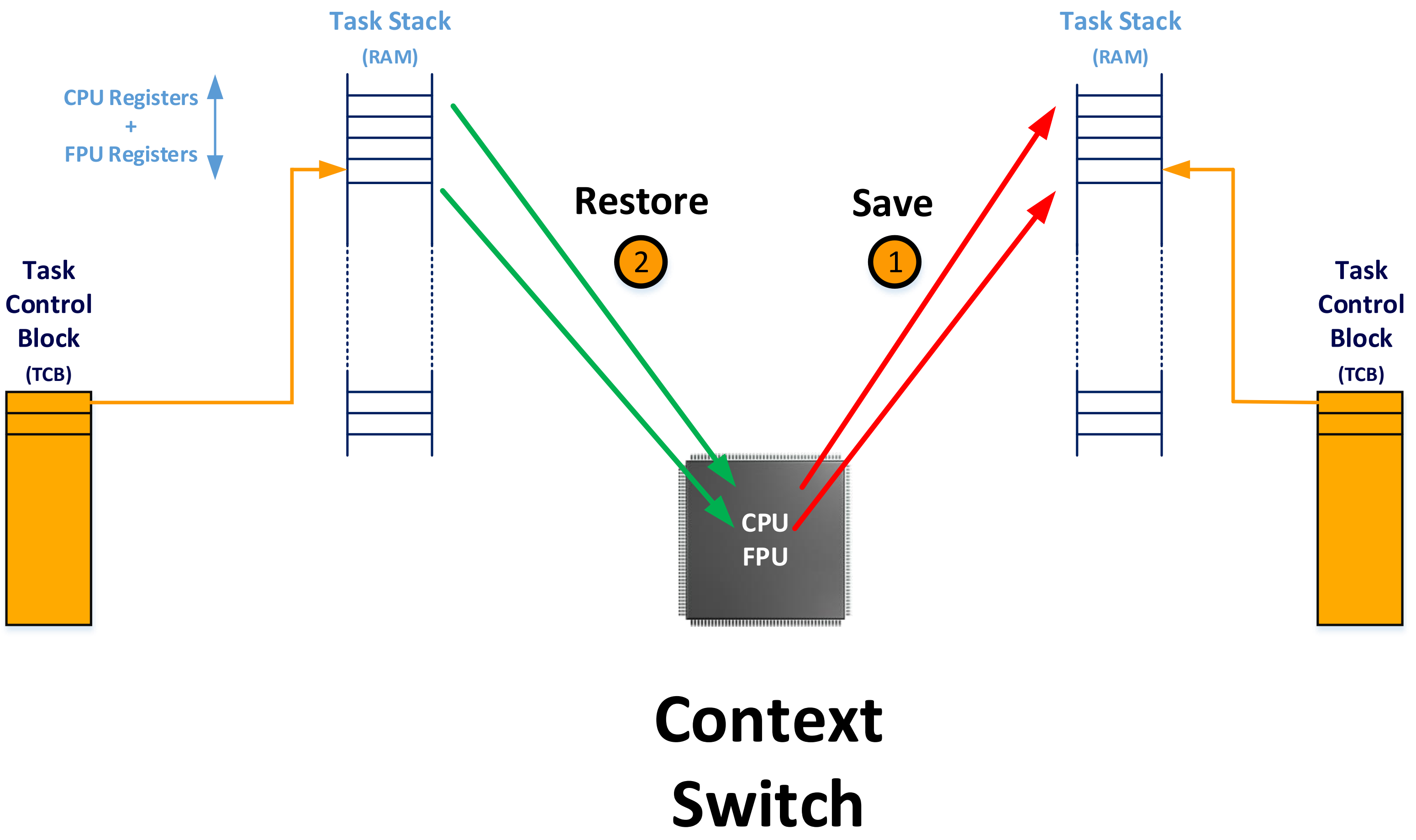
RTOS Basics – RTOS and User Code run in Privileged Mode

- Without an MPU, RTOS tasks run in Privileged mode
  - Access to all resources
  - Done for **performance** reasons

- **Drawbacks:**
  - Reliability of the system is in the hands of the application code
    - ISRs and tasks have **full** access to the memory address space
    - Tasks **can** disable interrupts
    - Task stacks can overflow **without** detection
    - Code **can** execute out of RAM
      - Susceptible to code injection attacks
    - A misbehaved task can take the whole system down
  - Expensive to get safety certification for the whole product



RTOS Basics – Context Switch (without an MPU)



Example using Cortex-M4

RTOSs are Event Driven

Type of Events

- **Data** available from another task

- From Kernel Aware Interrupts
  - Timer expires
  - DMA transfer completes
  - Ethernet packet arrives
  - etc.

- An ISR or a task signals another task
  - Through a semaphore
  - Through an event flag

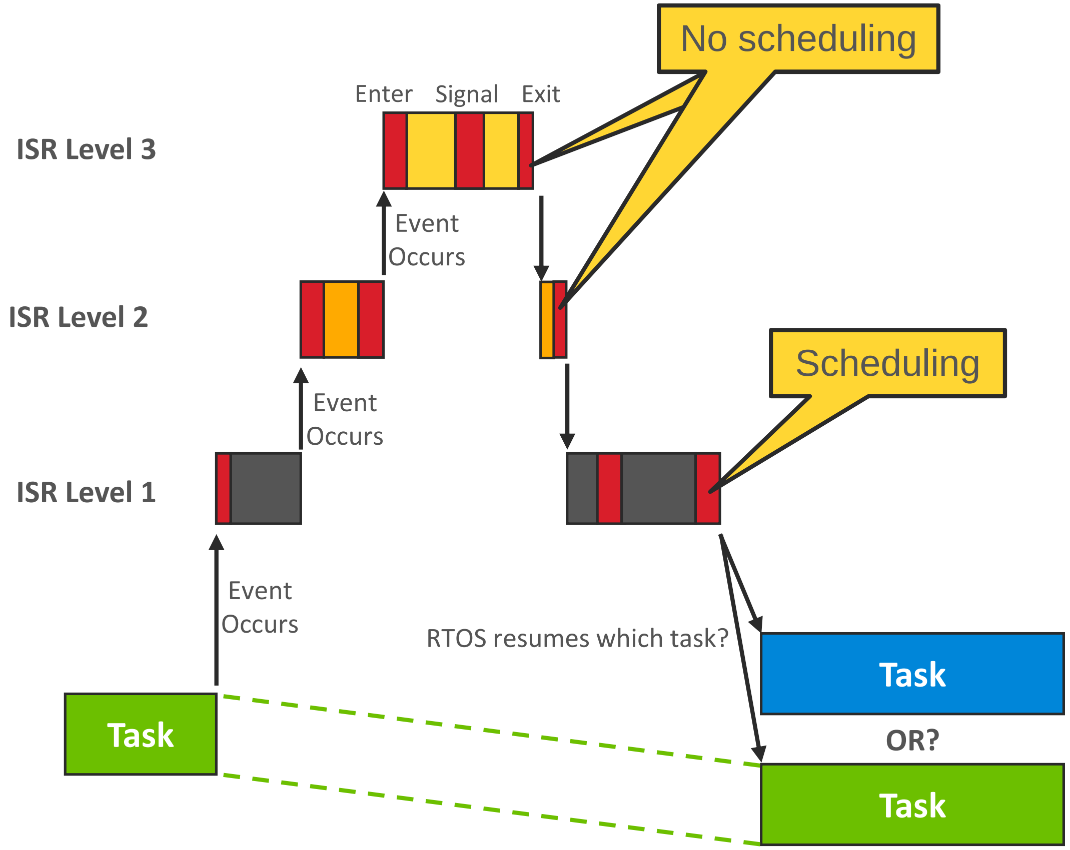
- A mutex is released

Kernel Aware Interrupt Events

- 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

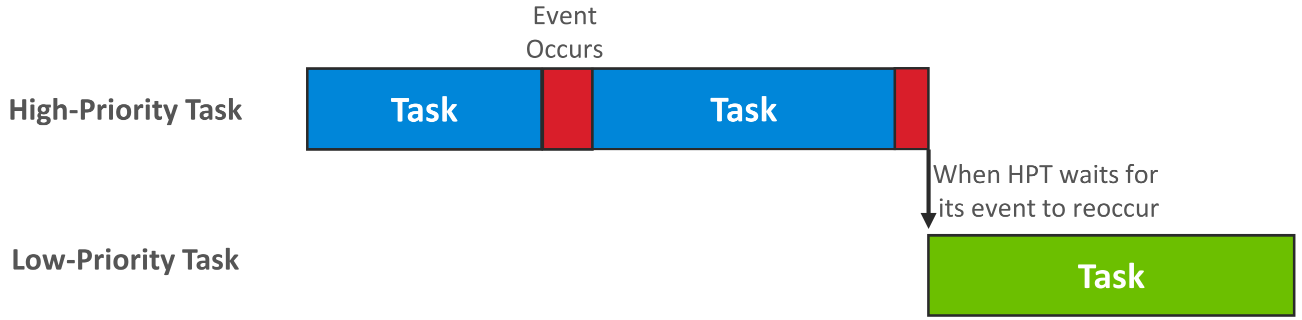
```
void MyISR (void)
{
    Entering ISR;
    :
    Signal or send a message to a MyTask;
    :
    Leaving ISR;
}
```

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

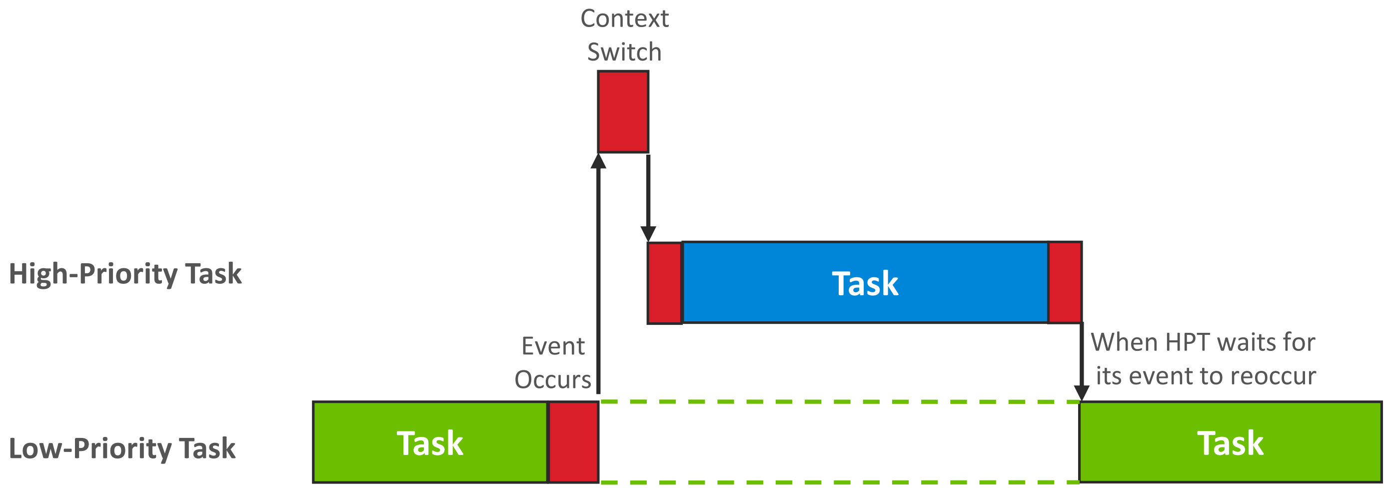


Tasks can also generate events for other tasks

- If a high-priority task generates an event that a low-priority task is waiting for, the high-priority task continues execution

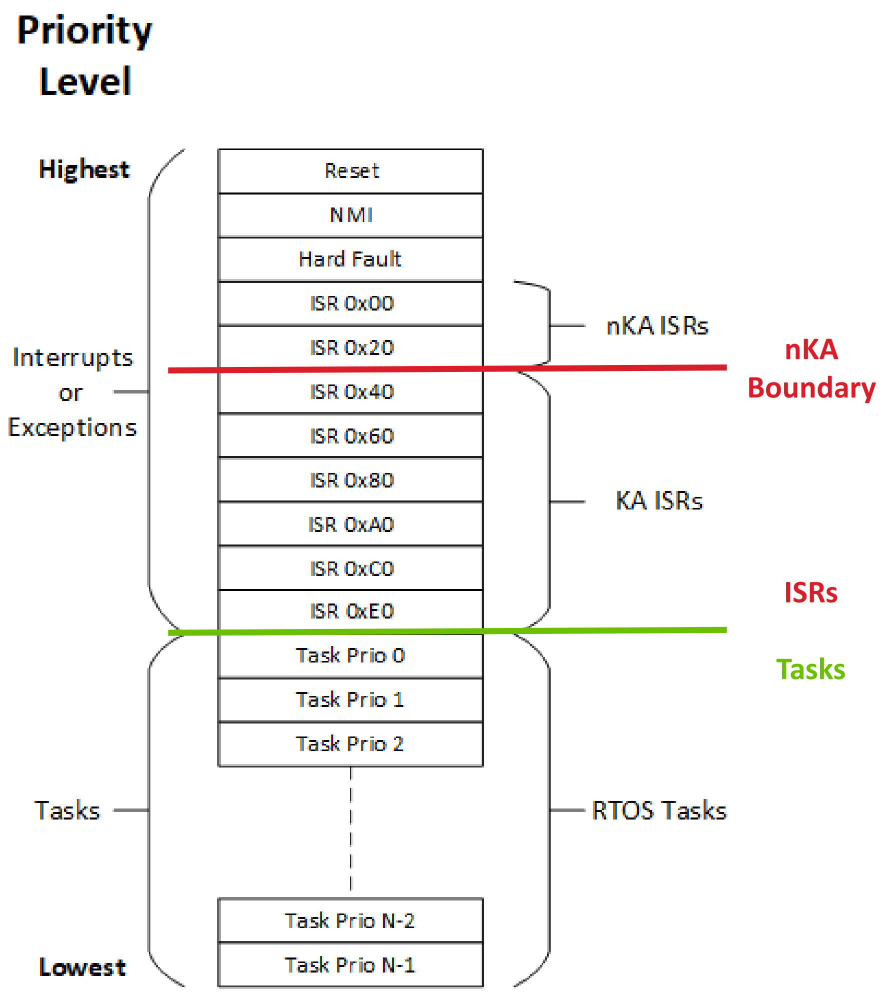


- If a low-priority task generates an event that a high-priority task is waiting for, the RTOS switches to the high-priority task



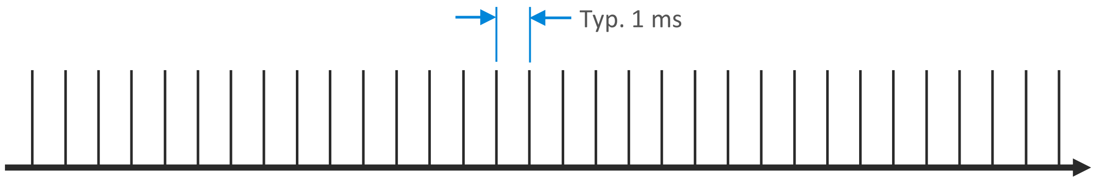
non-Kernel Aware Interrupts

- **Non-Kernel Aware (nKA)** ISRs
  - ISRs that have priorities higher than Kernel Aware ones
  - Your code **MUST NOT** make any RTOS API calls within these ISRs
  - Processors like the Cortex-M allow you to set the nKA boundary
- In order of priority:
  - Reset
  - NMI (Non-Maskable Interrupts)
  - nKA ISRs
  - KA ISRs
  - Highest priority task
  - Lowest priority task (typ. The RTOS's Idle Task)

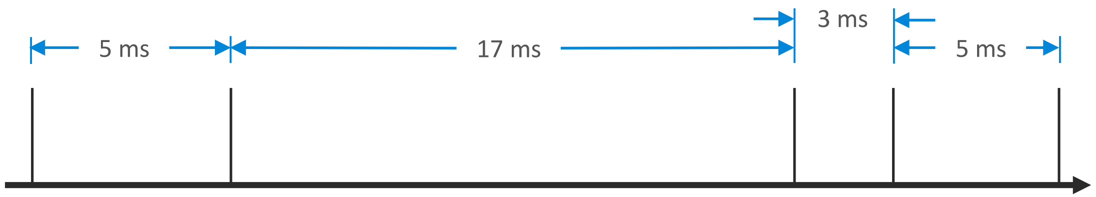


The Tick Interrupts – **Just** another source of Events!

- Most RTOS have a time-based interrupt
  - Called the ***System Tick*** or ***Clock Tick***
  - Requires a hardware timer
    - The Cortex-M has a dedicated RTOS timer called the ***SysTick***
- 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



Typical RTOS Tick

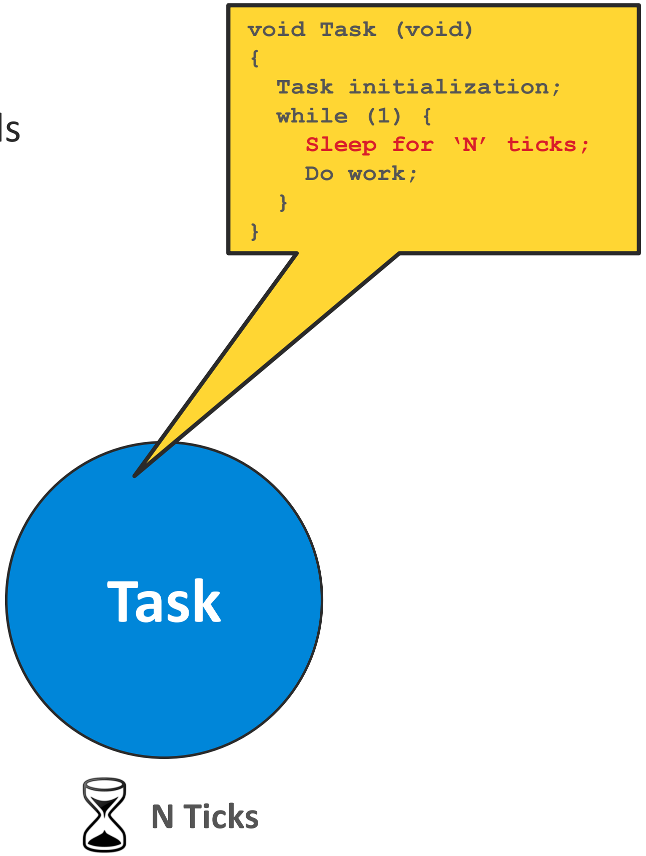


Dynamic RTOS Tick

RTOS Services

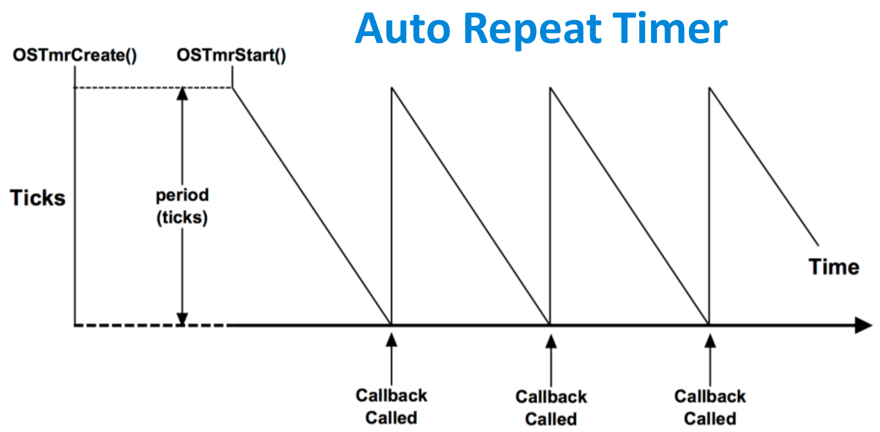
RTOS Services – Time Delays (i.e. Sleep)

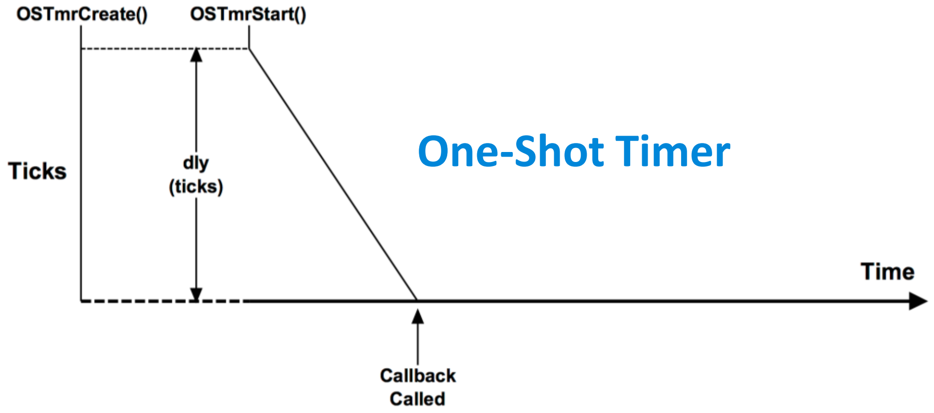
- A task can put itself to sleep by calling RTOS APIs:
  - **OSTimeDly()** // Delay for N ticks
  - **OSTimeDlyHMSM()** // Delay for Hours, Minutes, Seconds, Milliseconds
- Can be used to wake up a task at regular intervals
  - Control loops
  - Updating a display
  - Scanning a keyboard
  - Letting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



RTOS Services – Soft Timers

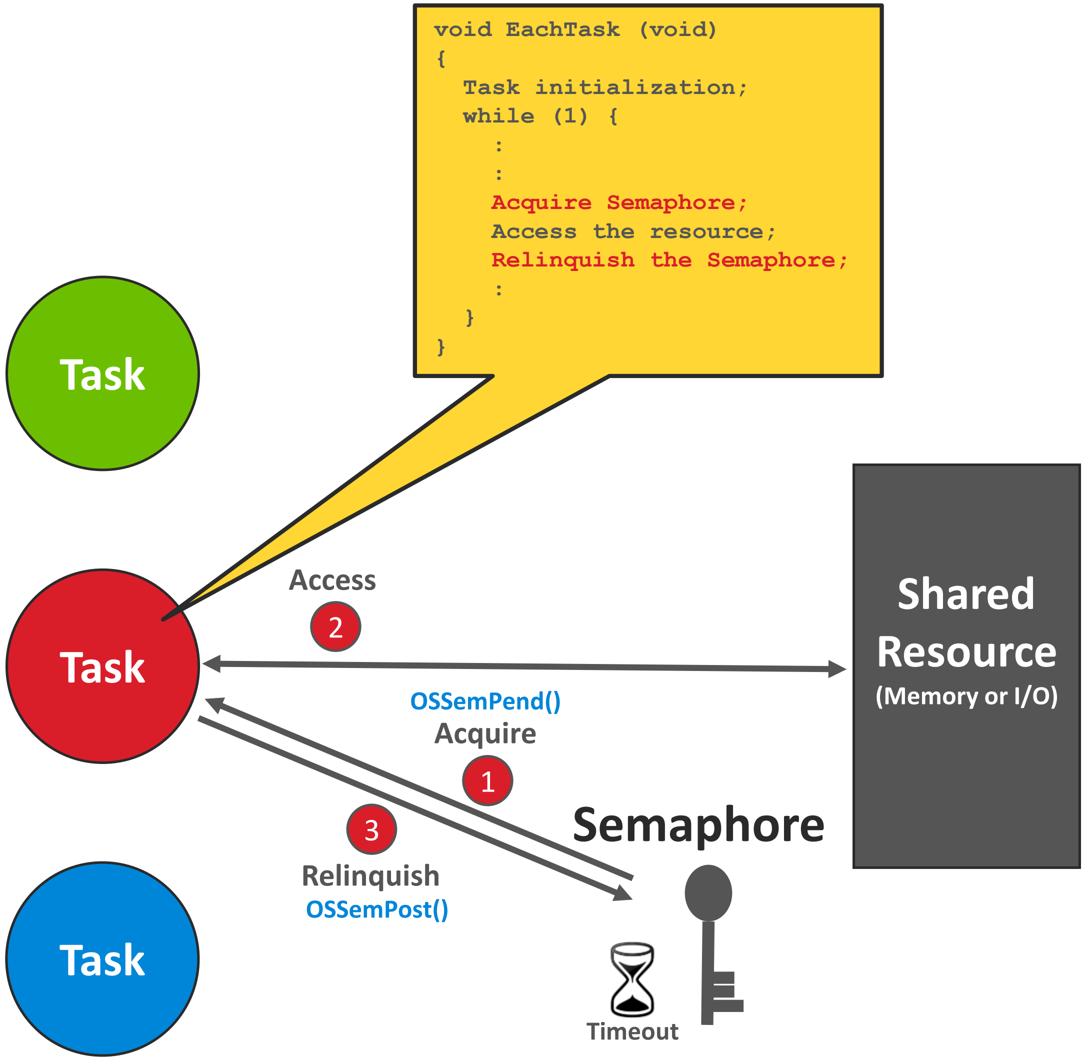
- Some RTOSs can provide soft timers which can be used to perform actions either once or at regular intervals
- A timer is an RTOS object containing:
  - An optional start **delay**
  - The amount of **time to expire**
  - A pointer to a **callback** to perform an action upon expiring
  - The option to auto repeat
- You can have an unlimited number of timers
  - Each timer must be **created** before it can be used
  - All of them execute in the context of a single task (i.e. the timer task)
- All timers are typically managed by an RTOS internal task
- Example usage:
  - Task opens a valve, starts a timer to close the valve after **X** seconds
  - Task starts a timer to blink a light



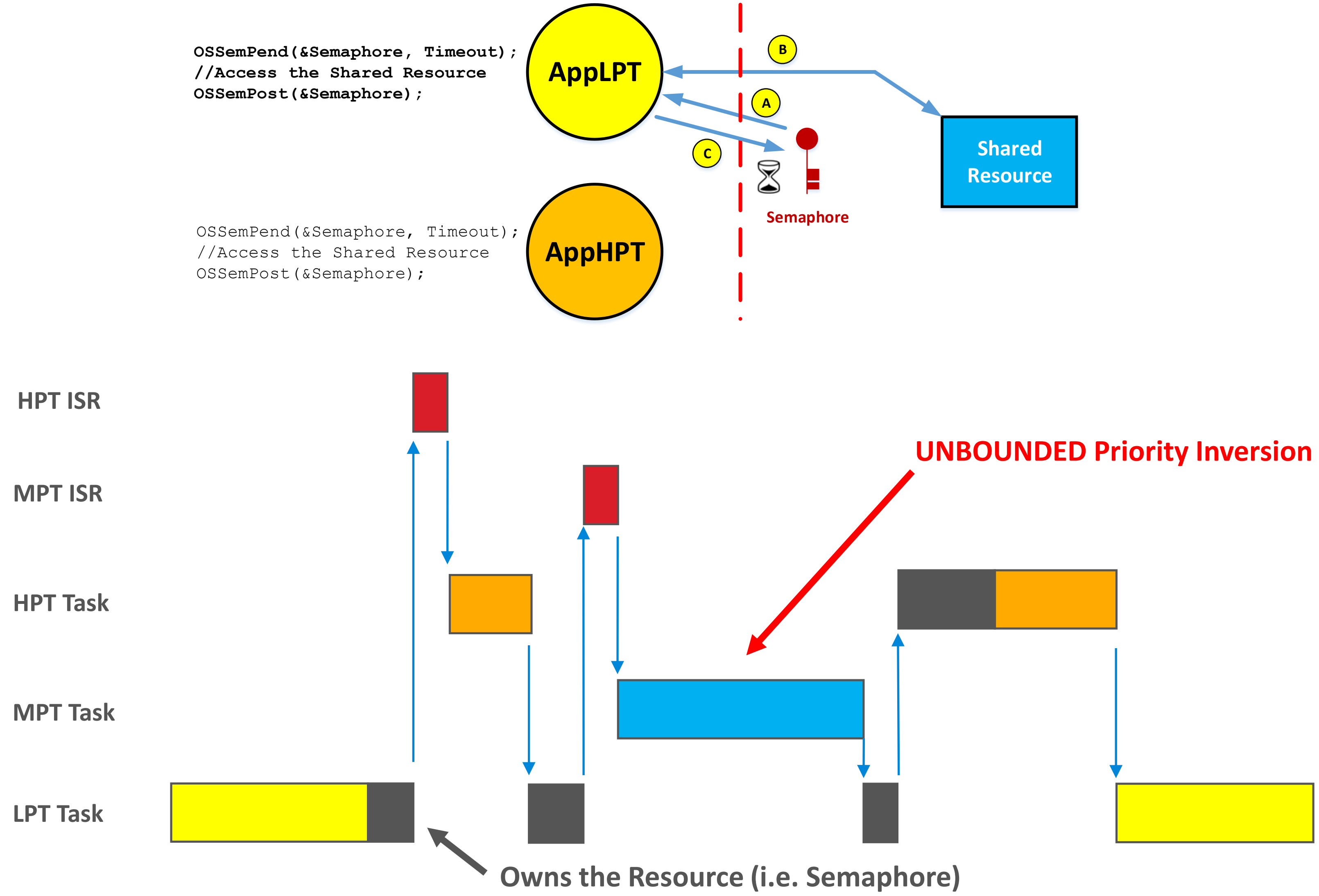


RTOS Services – Sharing A Resource – Using a **Semaphore**

- What is a resource?
  - Shared memory, variables, arrays, structures
  - I/O devices
- RTOSs **used** to use Semaphores for resource sharing
  - A Semaphore is an **RTOS object**
  - An semaphore must be **created** before it can be used
    - OSSemCreate()
  - Semaphores are subject to ***priority inversions …***

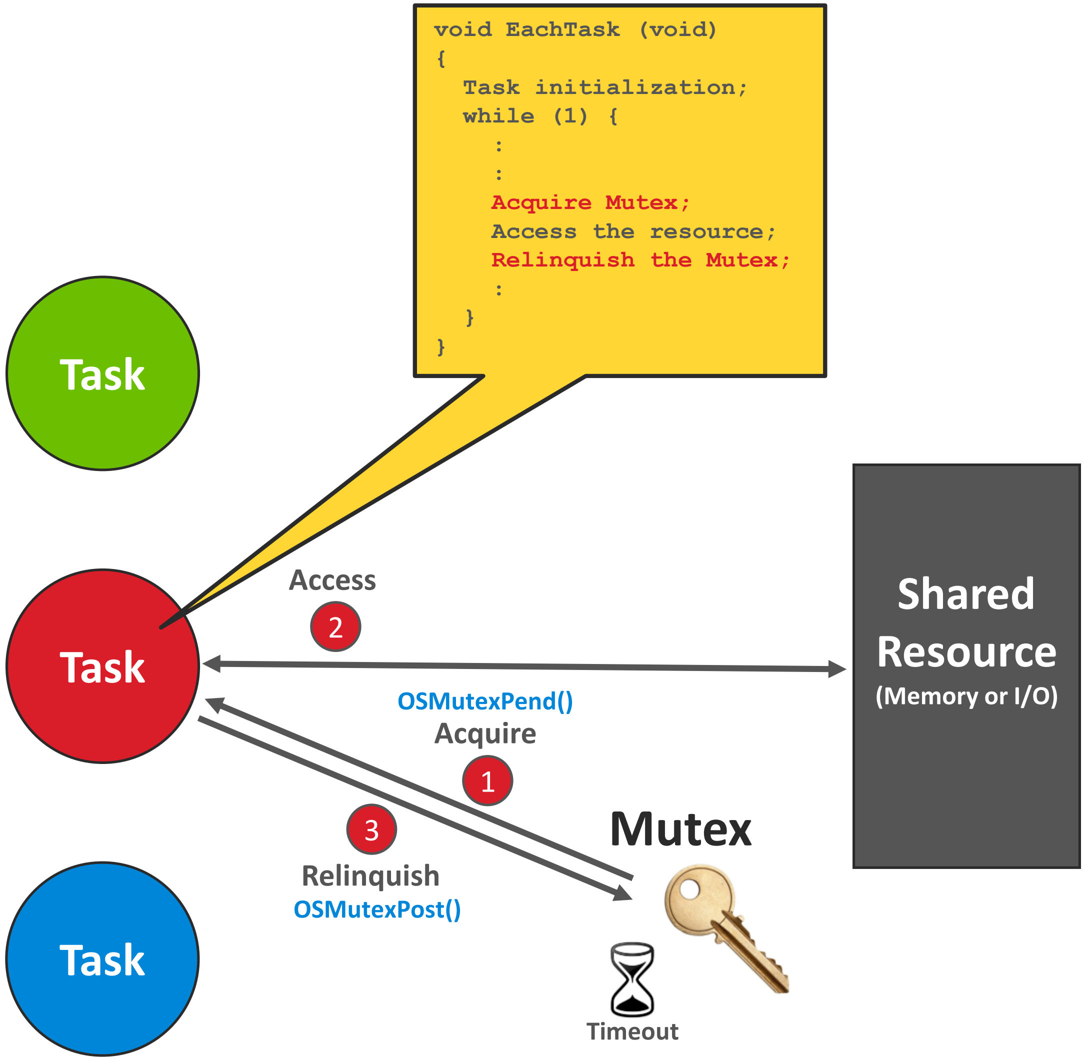


Priority Inversions Problem With Semaphores

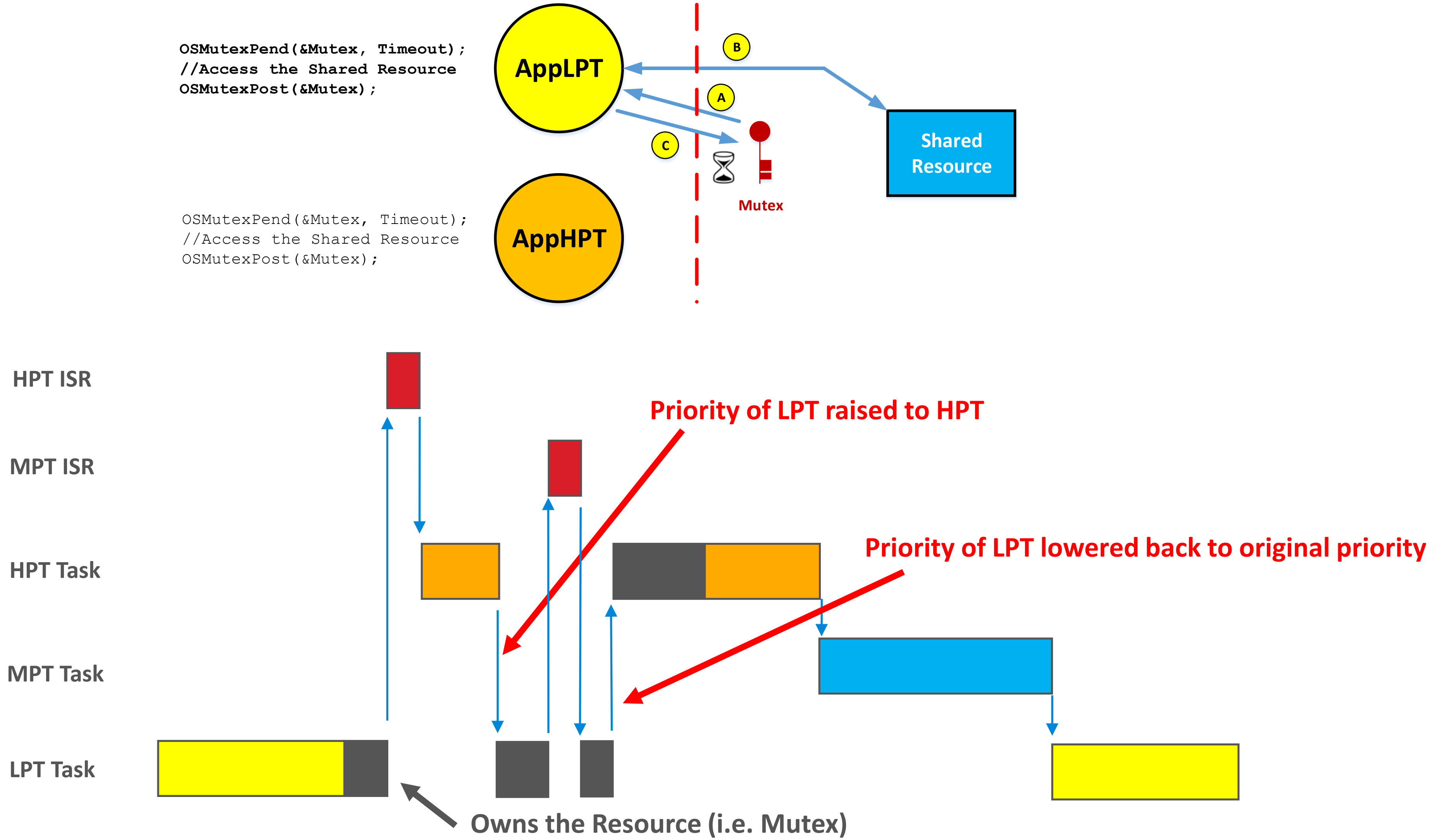


RTOS Services – Sharing A Resource

- RTOSs typically provide resource sharing APIs
  - Called ***Mutual Exclusion Semaphores*** (Mutex)
    - A Mutex is an **RTOS object** containing:
      - The key (binary value)
      - The priority of the mutex owner
      - A list of task waiting to acquire the mutex
  - An mutex must be **created** before it can be used
    - OSMutexCreate()
  - 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

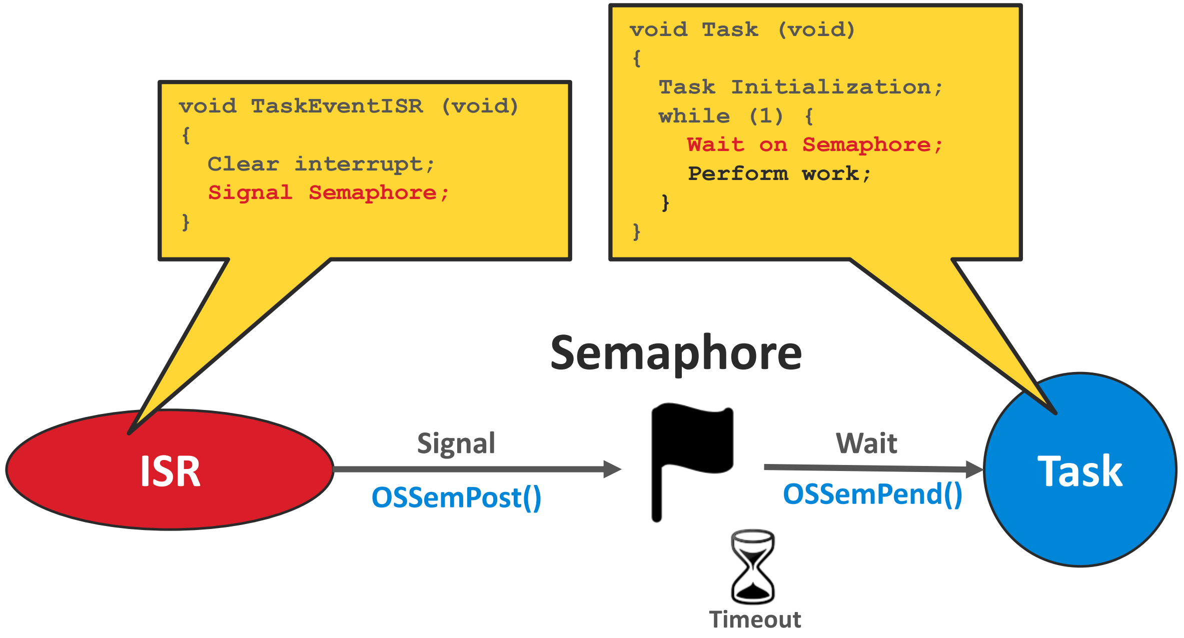


Unbounded Priority Inversion Avoided with Mutex

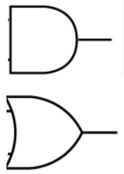


RTOS Services – Signaling A Task Using Semaphores

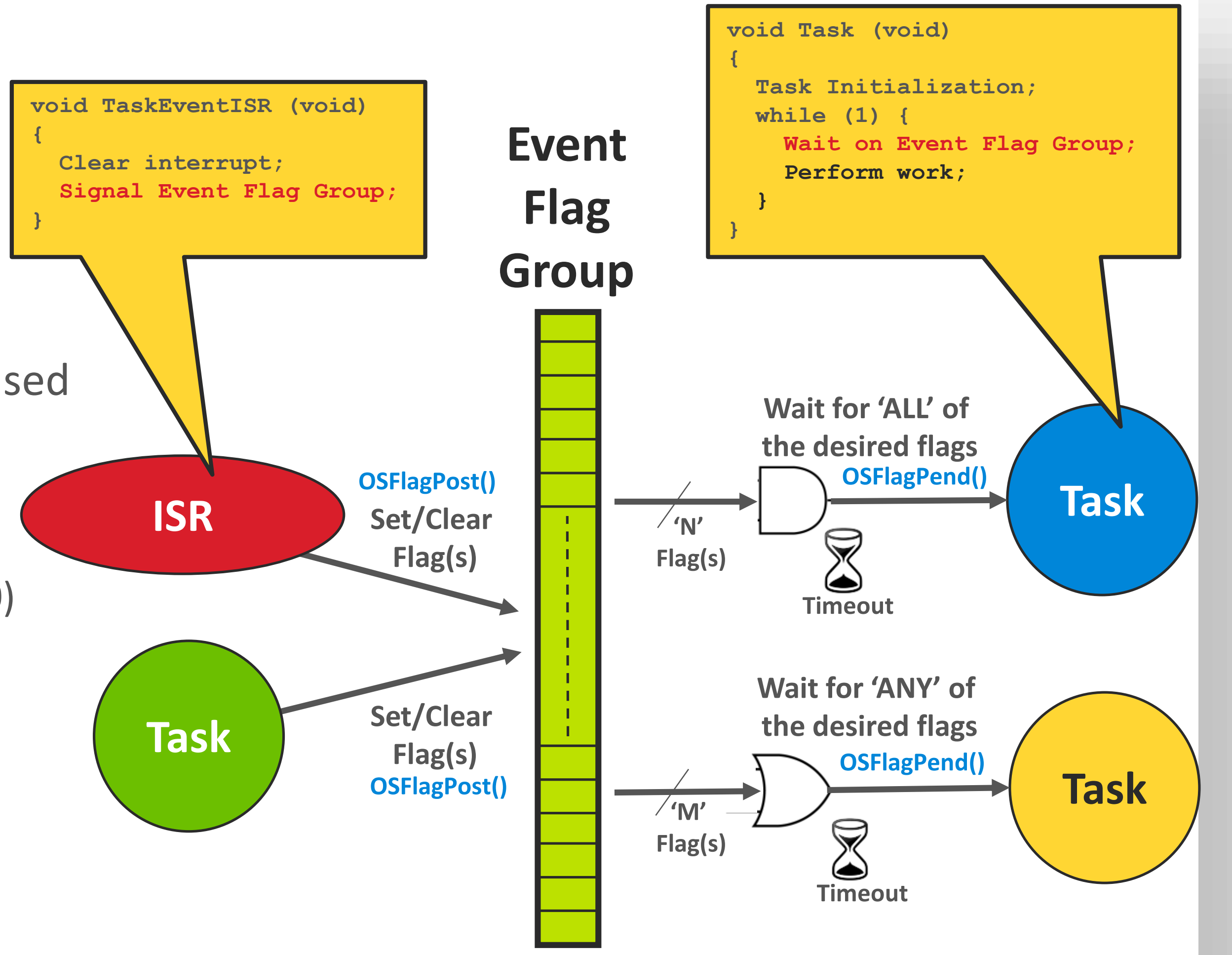
- Semaphores can be used to **signal** a task
  - Called from ISR or Task
  - Does not contain data
- A Semaphore is an **RTOS object** containing:
  - A counter to accumulate unprocessed signals
  - A list of tasks waiting for the event to occur
    - Typically only 1 task waits on a given semaphore
- An semaphore must be **created** before it can be used
  - **OSSemCreate()**



RTOS Services – Signaling Task(s) Using Event Flags

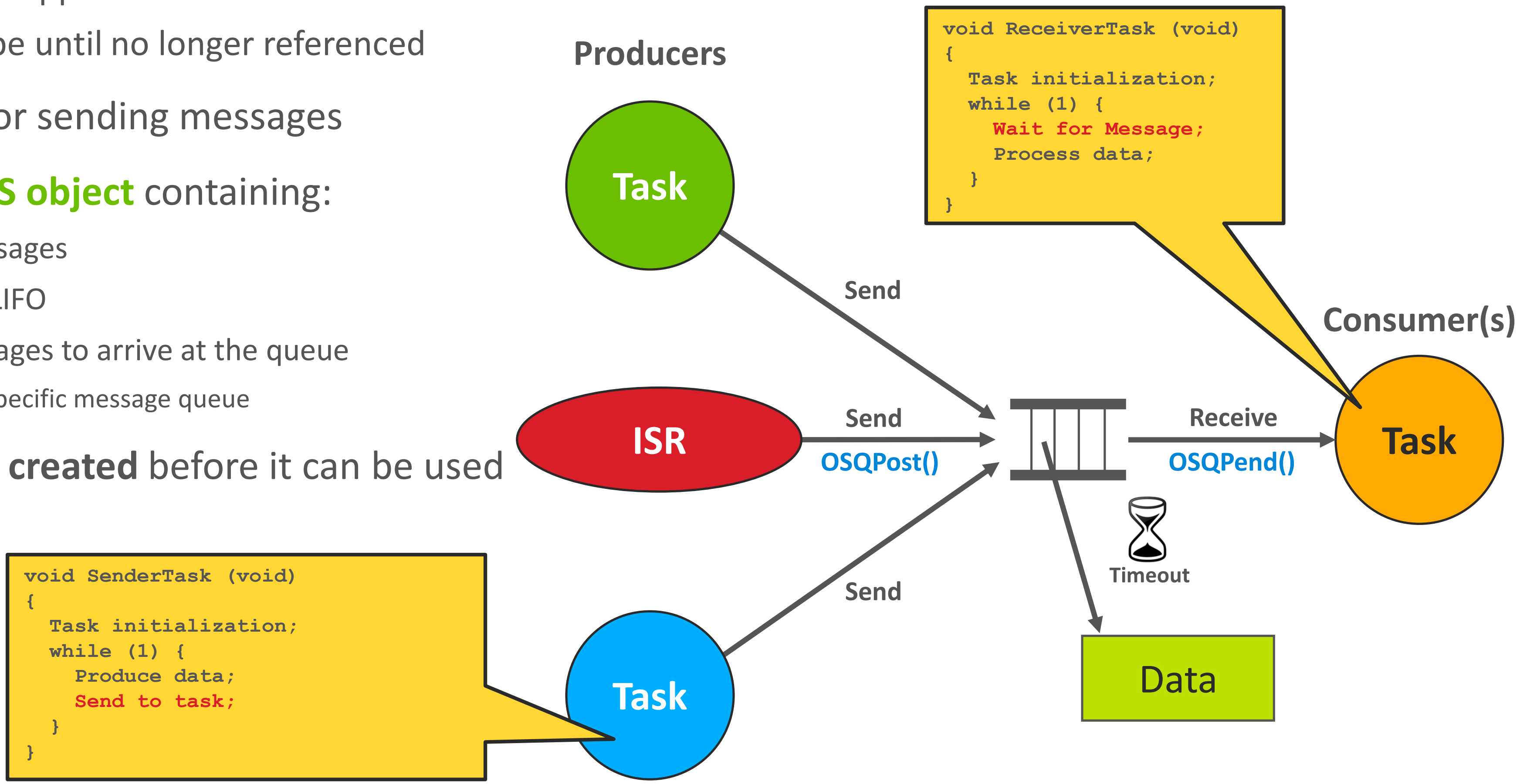


- **Event Flags** are a grouping of bits used to signal the occurrence of more than one events
  - Signals from ISRs or Tasks
  - Only tasks can wait for events
  - Does not contain data (just happened or not)
- An Event Flag group must be **created** before it can be used
  - OSFlagCreate()
- A Event Flag group is an **RTOS object** containing:
  - The **current state** of each of the N-bits in a group (i.e. 1 or 0)
    - Each corresponds to an **event**
    - Typically 8, 16 or **32** bits per group
  - A list of tasks waiting on the Event Flag group
    - Each task waits for **desired** bit (**OR**-condition or **AND**-condition)



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
- Message queues are used for sending messages
- A message queue is an **RTOS object** containing:
  - A queue that can hold 'N' messages
  - Queues can either be FIFO or LIFO
  - A list of tasks waiting for messages to arrive at the queue
    - Typically only 1 task waits on a specific message queue
- An message queue must be **created** before it can be used
  - OSQCreate()

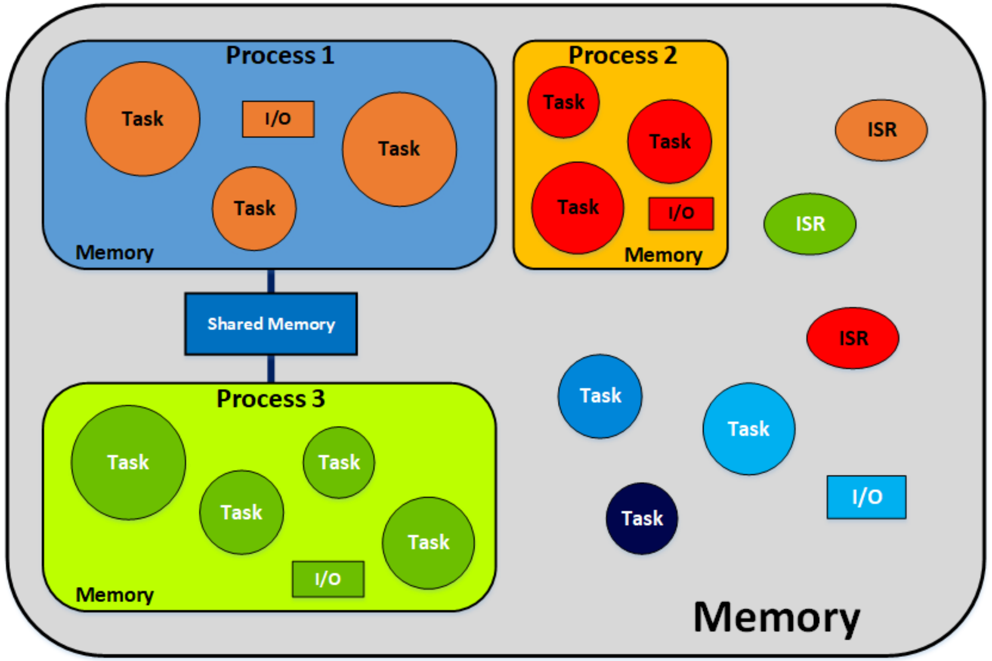


Quick Break - ~15 Minutes

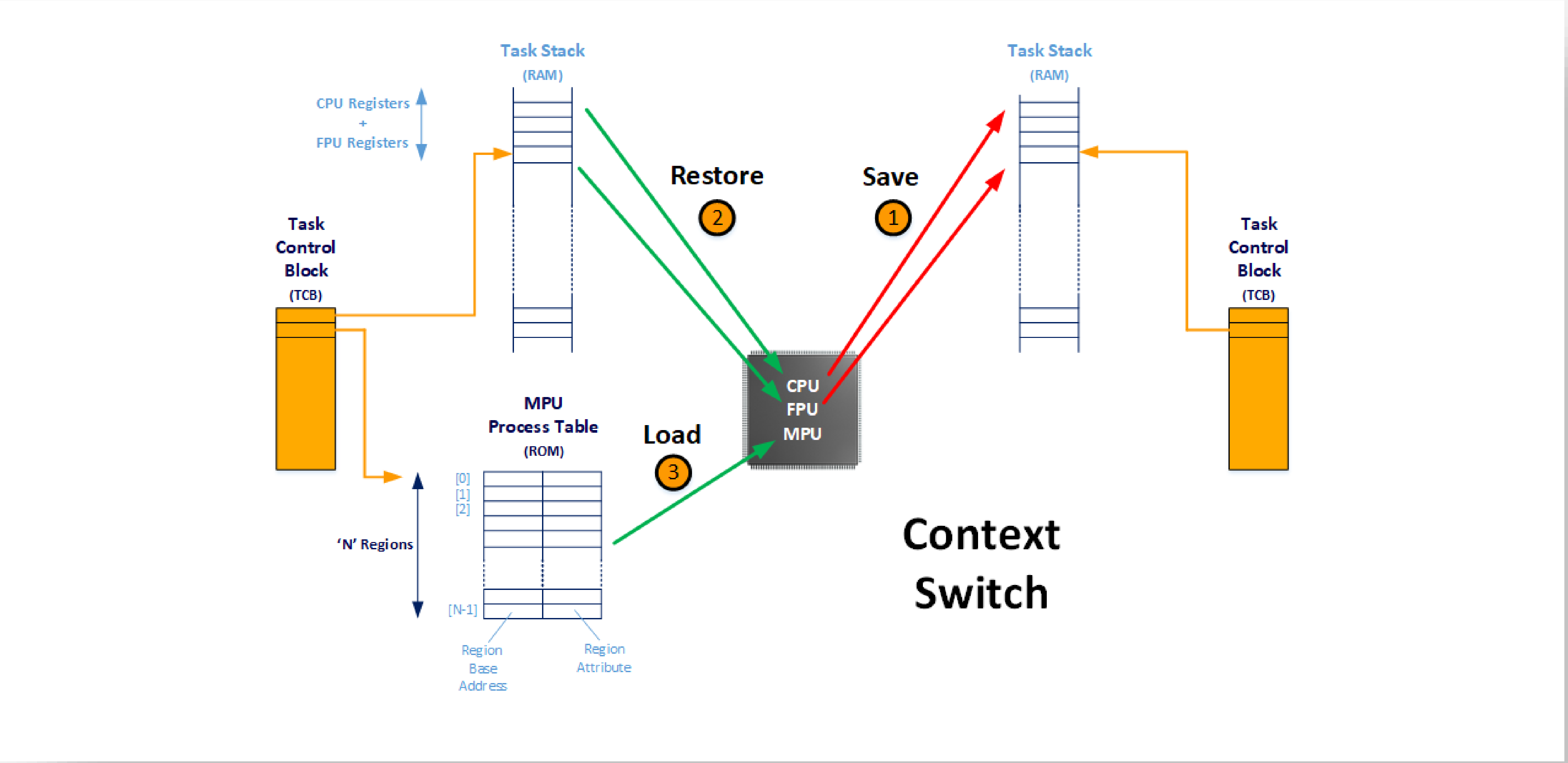
Process Separation

Process Separation – Process Model (Requires an MPU or MMU)

- Tasks are **grouped** by processes
  - Can have **multiple** tasks per process
  - Memory of one process is **not accessible** to other processes
    - Unless they **share** a common memory space
- ISRs typically have **full** access to memory
  - Would be **very** complex otherwise
- I'll assume a Cortex-M MPU from now on
- User tasks **can't** disable/enable interrupts
  - Also **cannot** alter the interrupt controller settings
  - This is a P/NP feature, not an MPU one
    - Requires an SVC handler
- Task stack overflows can be detected with the MPU
  - Not needed for **ARMv8-M** because of stack limit registers
- MPU configuration consist of setting up a ***process table*** for each task

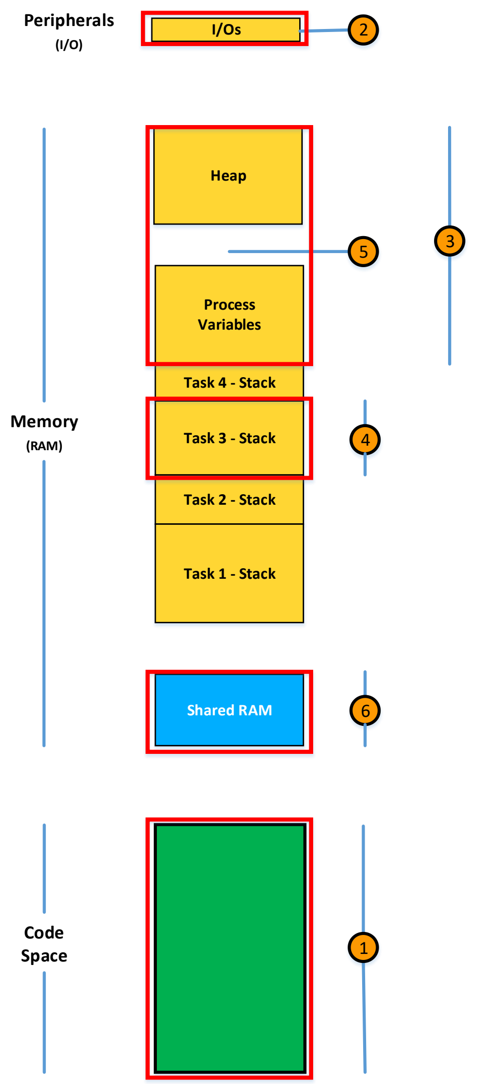


Process Separation – Context Switch

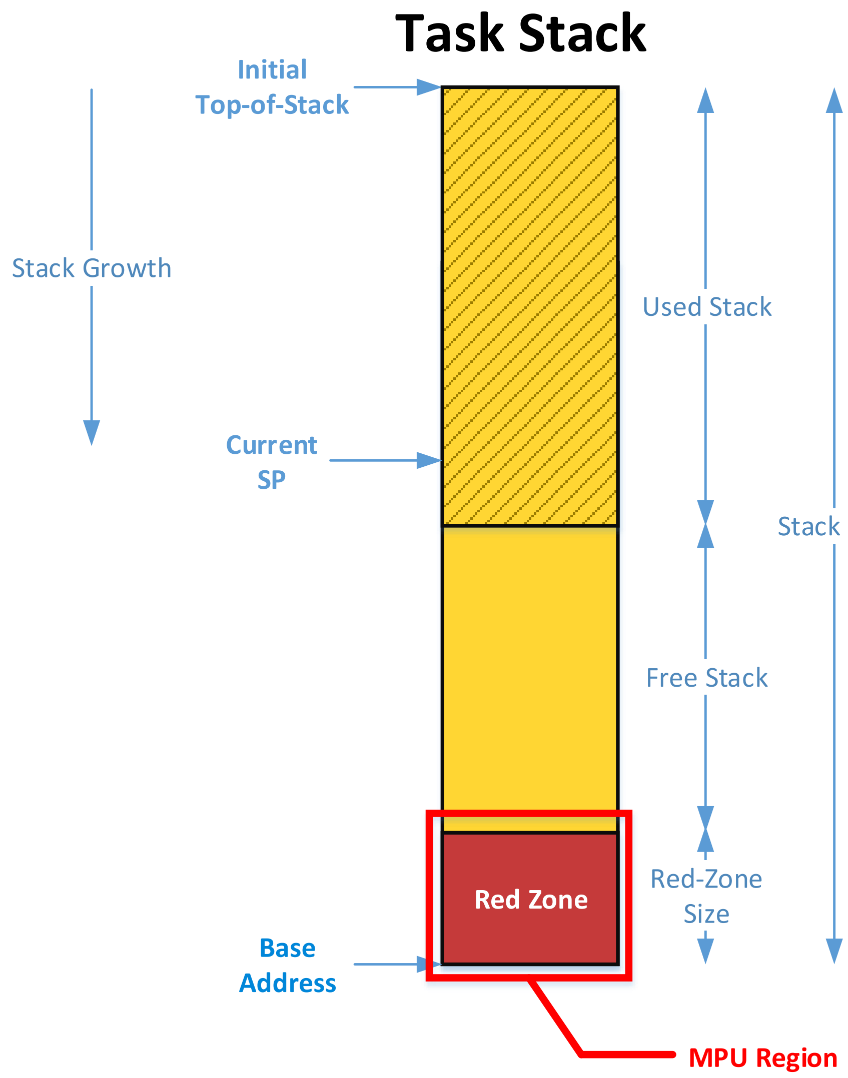


Process Separation – Expanded Process View

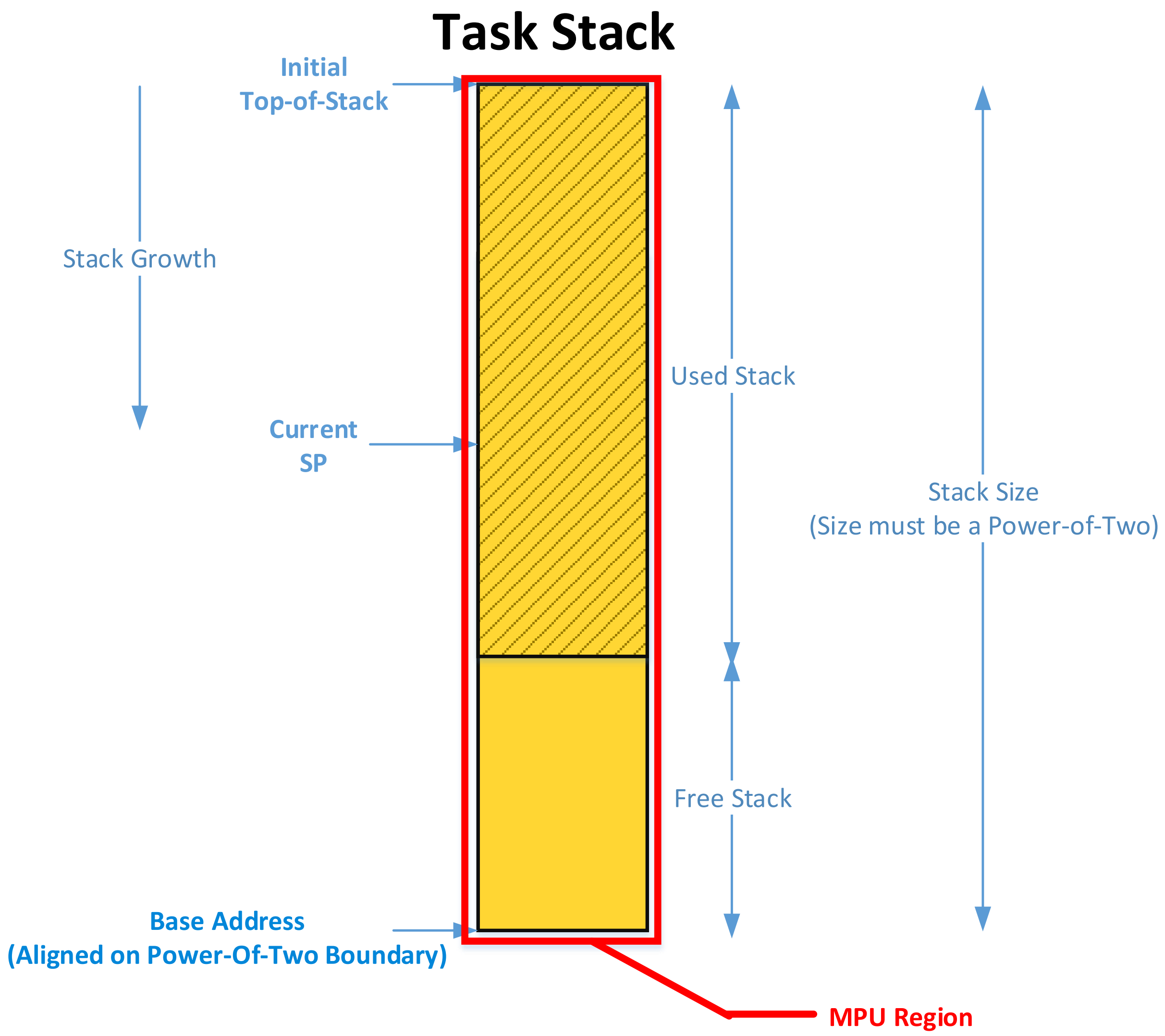
- A task can have up to **8** or **16** regions
- (1) Full access to code space
  - Typically don't limit access to code
- (2) At least one region for process peripheral
  - May need more than one
- (3) One region to access the RAM for the process
  - On **ARMv7-M**, size must be a power of 2
  - On **ARMv8-M**, size *doesn't have to be* a power of 2
- (4) One region stack overflow detection
  - … see next slide
  - Not needed for **ARMv8-M**
- (5) This is unused area
  - On **ARMv8-M**, this can be as small as 32 bytes
- (6) Memory to be shared with other processes
  - If needed



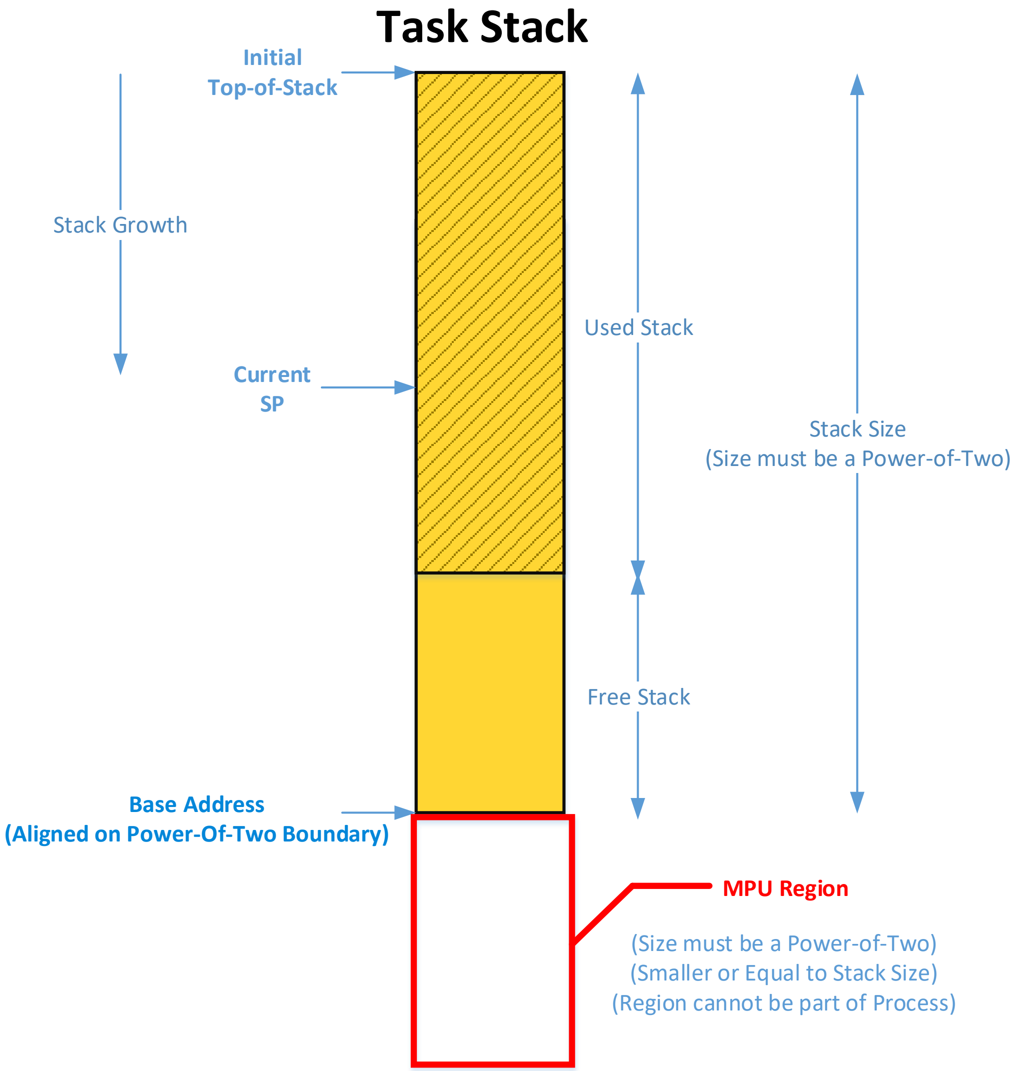
Process Separation – Stack overflow detection – Method #1



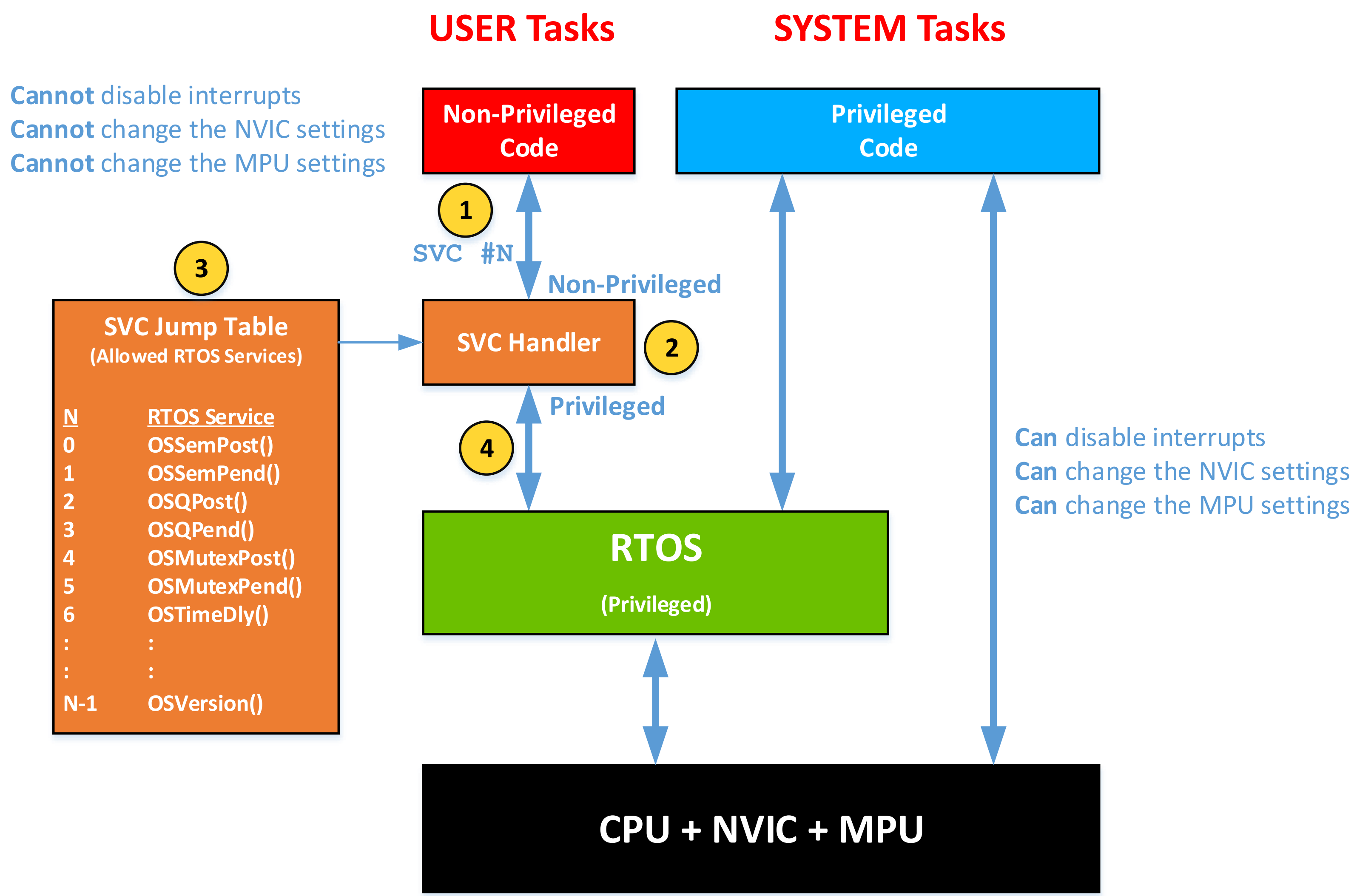
Process Separation – Stack overflow detection – Method #2



Process Separation – Stack overflow detection – Method #3



Process Separation – User tasks run in Non-Privileged mode

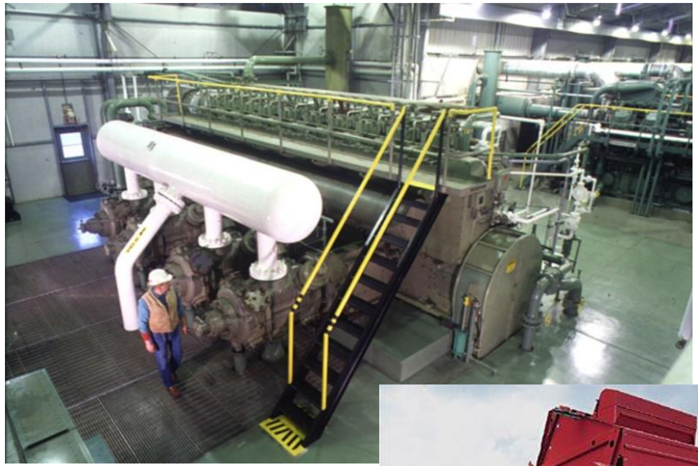


Process Separation – Handling Faults

- What happens when a task accesses data outside a valid region?
  - The MPU issues an exception called the *MemManage* Fault
- What can we do when a fault is detected?
  - Depends greatly on the application
  - The RTOS should save information about the offending task
    - To help developers correct the problem
  - The RTOS should provide a callback function for each task
    - To allow the application to perform a *Controlled Shutdown* sequence
      - Actuators to be placed in a safe state
  - Terminate the offending task?
    - Do we also need to terminate other tasks associated with the process?
    - What happens to the resources owned by the task(s)?

Seeing Inside Live Embedded Systems

Debugging Live Systems



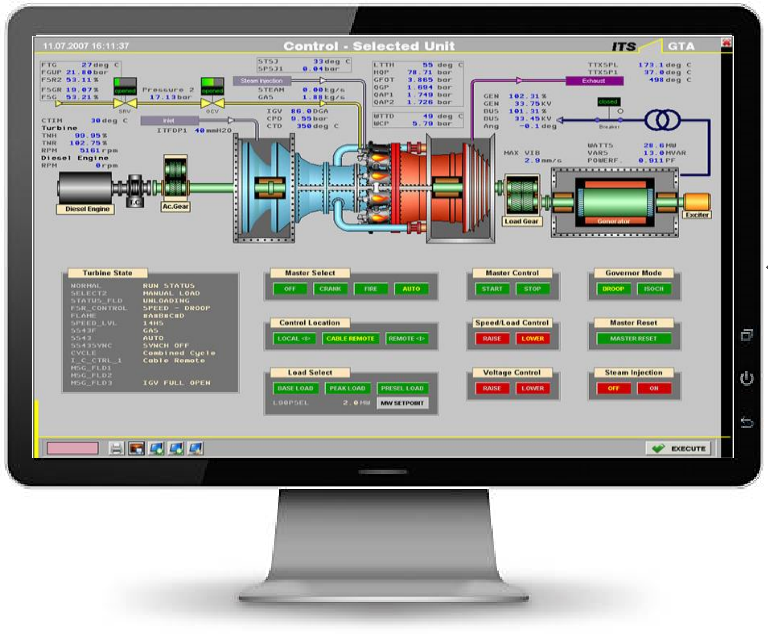


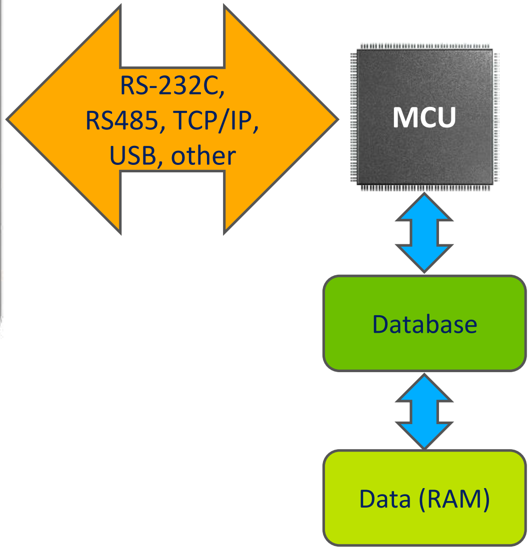
- You can’t always ‘single step’ through code!
  - Engine control
  - Printing presses
  - Food processing
  - Flight management
  - Chemical reactions
  - Agricultural equipment
  - Etc.
- Stopping these systems can have disastrous and/or costly consequences
  - Must be tested and debugged live

How Do You ‘See’ Inside These Systems?

- Displaying values using:
  - LED annunciators
  - 7-Segment numeric displays
  - Bar graphs
  - Alphanumeric displays
  - Graphical user interfaces (GUIs)
  - `printf()` statements to a terminal
  - Debugger’s live watch … limited to numerical values
  - Etc.
- Drawbacks:
  - Display capabilities might be limited
  - All require target resident code
    - Heisenberg effect is often significant
  - Limited to what you can see/change
  - If you forget something …
    - Rebuild code
    - Download
    - Try to get back to the same test conditions

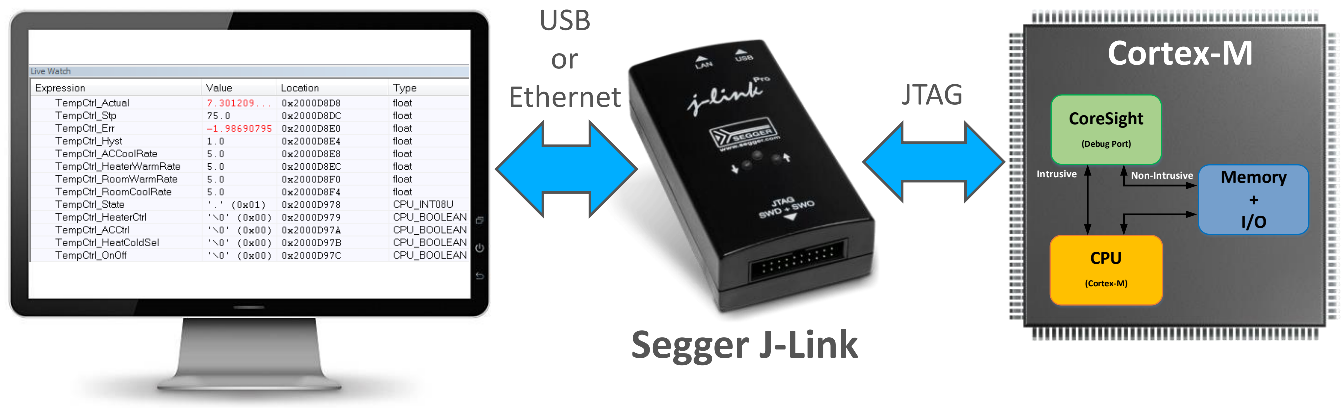
What if We Move the Display/Controls to a PC?





- Using COTS man-machine interfaces (MMIs)
  - e.g. Wonderware ‘InTouch’ (Schneider)
  - Much better at visualizing the process
  - Can monitor and/or change hundreds of values
  - Data logging capabilities
- Uses standard PLC protocols
  - e.g. Modbus, ProfiNet, DeviceNet, etc.
- Drawbacks:
  - Target needs a database of accessible variables
  - Requires target resident code
  - Adds overhead, complexity and cost
- COTS MMIs are typically for end use
  - Could be useful during development

Debugging RTOS-Based Systems – ARM CoreSight Debug Port



- Core debugging:
  - Halting
  - Single stepping
  - Resume
  - Reset
  - Register accesses
- Up to 8 hardware breakpoints
- Up to 4 hardware watchpoints
- Optional ***instruction*** trace
- ***Data*** trace
- Instrumentation trace (printf() like) – 32 ch
- Profiling counters
- PC sampling
- **On-the-fly memory and I/O accesses**
  - Can be a security risk for deployed systems though

Debugger Live Watch

```c
static  void  AppTempCtrl (void)
{
    AppTempErr      = AppTempActual - AppTempStp;
    AppTempHeatRate = ((CPU_FP32)AppTempHeaterWatts / (CPU_FP32)1000.0)
                    * ((CPU_FP32)1.0 / (CPU_FP32)AppTempRoomSize);
    AppTempCoolRate = ((CPU_FP32)1.0 / (CPU_FP32)AppTempRoomSize);
    if (AppTempActual > (AppTempStp + AppTempHyst)) {      /* Determine what state we are in            */
        AppTempState    = 3;                               /* Above Stp + Hyst                          */
    } else if (AppTempActual < (AppTempStp - AppTempHyst)) {
        AppTempState    = 1;                               /* Below Stp - Hyst                          */
    } else {
        AppTempState    = 2;                               /* Between Stp + Hyst and Stp - Hyst         */
    }

    if (AppTempCtrlEn == DEF_ENABLED) {                    /* See if controller is turned on            */
        BSP_LED_Toggle(2);
                                                           /* ------------ HEATING MODE --------------- */
        if (AppTempSelHeat == DEF_ON) {                    /* See if heater is selected                 */
            AppTempAC_Ctrl     = DEF_OFF;
            switch (AppTempState) {
                case 1:
                     AppTempHeater_Ctrl = DEF_ON;
                     AppTempActual      += AppTempHeatRate;
                     BSP_LED_On(3);
                     BSP_LED_Off(1);
                     break;

                case 2:
                     if (AppTempHeater_Ctrl) {
                         AppTempActual += AppTempHeatRate;
                     } else {
                         AppTempActual -= (CPU_FP32)0.0005;    /* Cool the room at natural rate         */
                     }
                     break;

                case 3:
                     AppTempHeater_Ctrl = DEF_OFF;
                     AppTempActual      -= (CPU_FP32)0.0005;  /* Cool the room at natural rate          */
                     BSP_LED_Off(3);
                     BSP_LED_Off(1);
                     break;
            }
                                                               /* ---------- COOLING MODE ------------- */
        } else {                                               /* We want to get the room colder        */
            AppTempHeater_Ctrl = DEF_OFF;
            switch (AppTempState) {
```

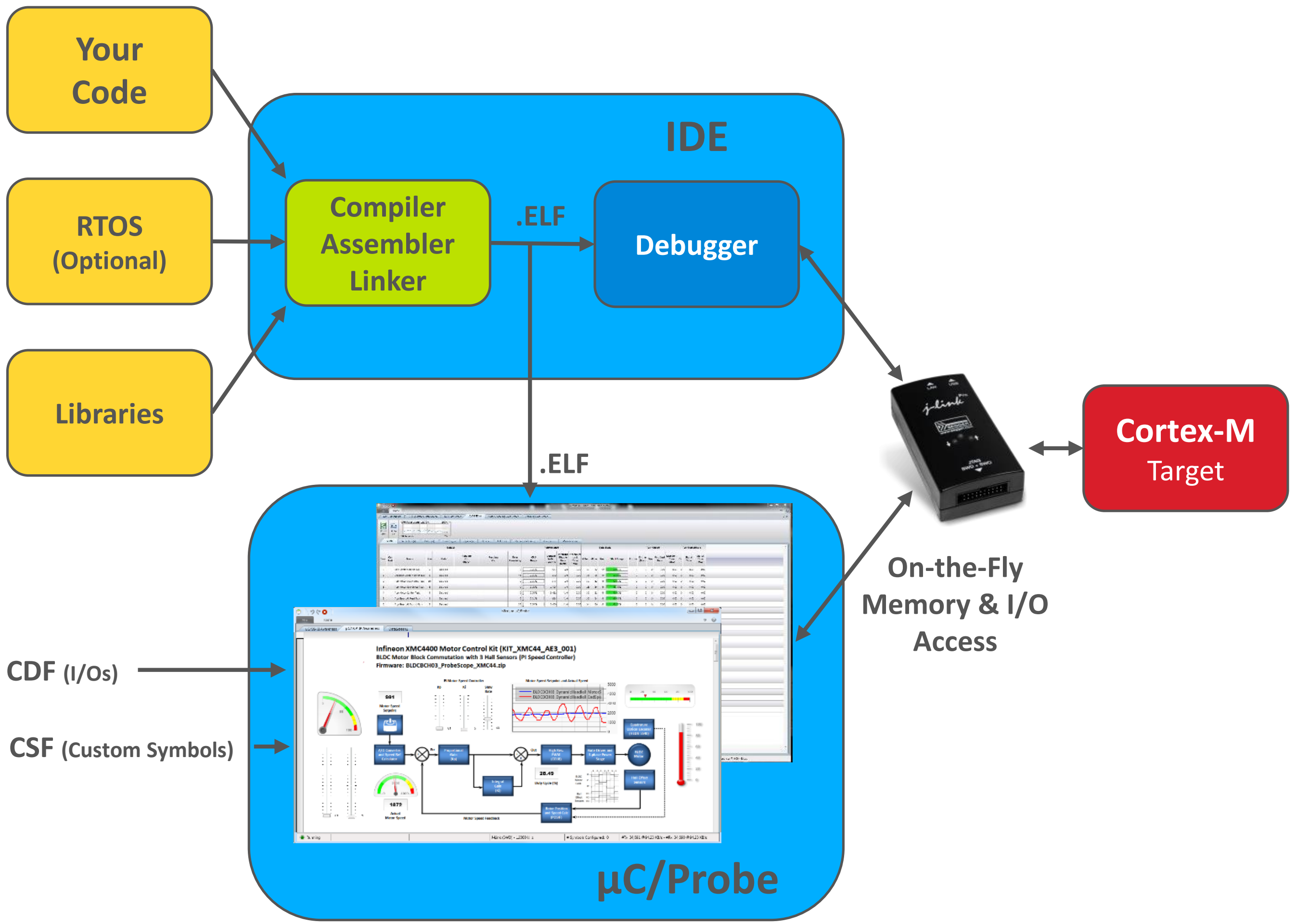
- Debuggers have offered ***Live Watch*** for years
  - Uses the on-the-fly-feature of the Cortex-M
- Typically only displays numerical values
  - Difficult to see trends and orders of magnitudes
  - Choice of Decimal, Hex, Float, etc.
- Update rate is typically 1 Hz

Live Watch

Expression	Value	Location	Type
TempCtrl_Actual	7.301209...	0x2000D8D8	float
TempCtrl_Stp	75.0	0x2000D8DC	float
TempCtrl_Err	-1.98690795	0x2000D8E0	float
TempCtrl_Hyst	1.0	0x2000D8E4	float
TempCtrl_ACCoolRate	5.0	0x2000D8E8	float
TempCtrl_HeaterWarmRate	5.0	0x2000D8EC	float
TempCtrl_RoomWarmRate	5.0	0x2000D8F0	float
TempCtrl_RoomCoolRate	5.0	0x2000D8F4	float
TempCtrl_State	'.' (0x01)	0x2000D978	CPU_INT08U
TempCtrl_HeaterCtrl	'\0' (0x00)	0x2000D979	CPU_BOOLEAN
TempCtrl_ACCtrl	'\0' (0x00)	0x2000D97A	CPU_BOOLEAN
TempCtrl_HeatColdSel	'\0' (0x00)	0x2000D97B	CPU_BOOLEAN
TempCtrl_OnOff	'\0' (0x00)	0x2000D97C	CPU_BOOLEAN

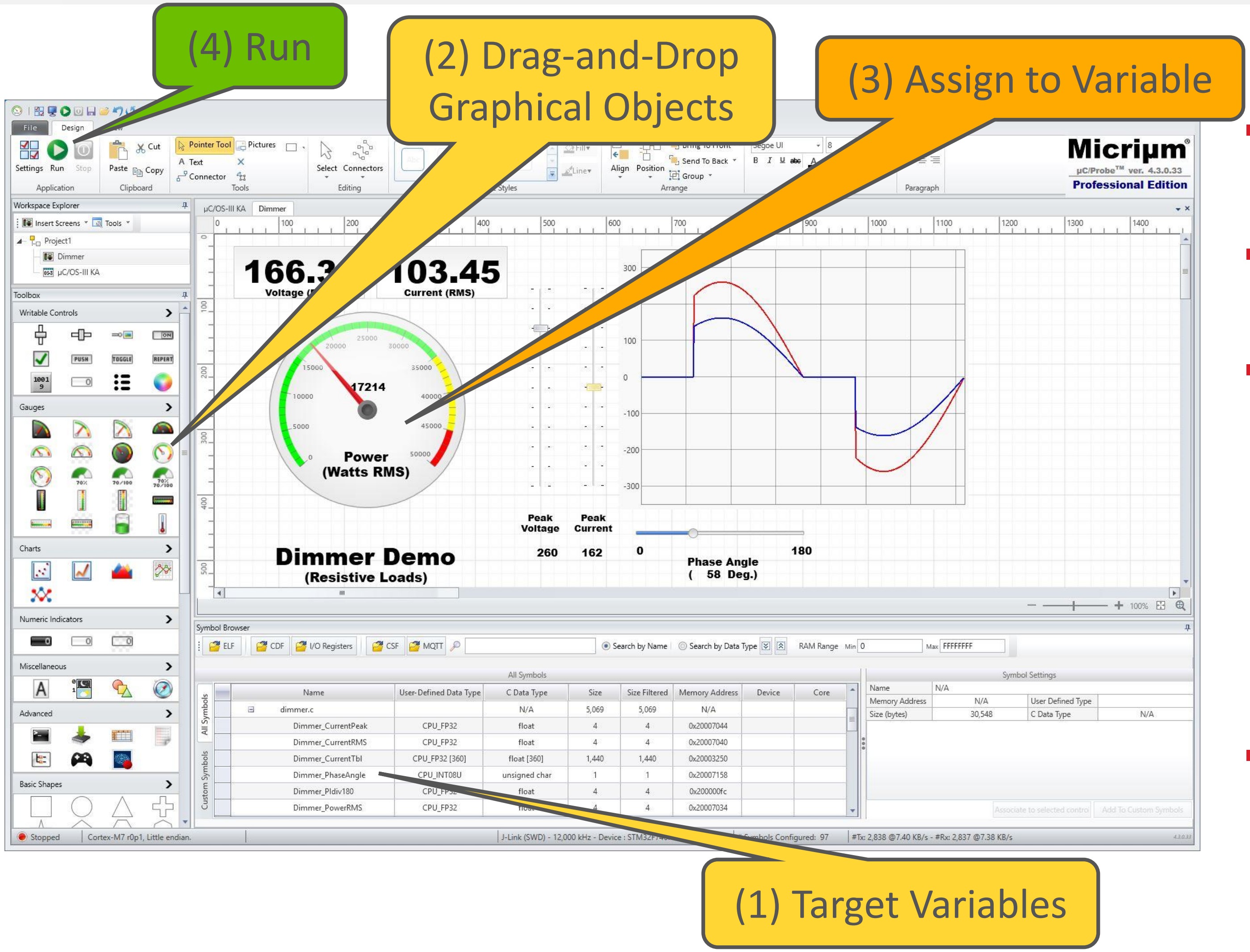
Micrium's µC/Probe, Graphical Live Watch®
(www.micrium.com)

µC/Probe, Graphical Live Watch®



- µC/Probe is an MMI for embedded systems
  - Use the **.ELF** as the database (same as downloaded code)
  - Like a doctor's stethoscope (non-intrusive)
- Adding ***graphics*** capabilities to ***Live Watch***
  - Display or change 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 or USB
- For **bare metal** or **RTOS**-Based applications
  - **Micrium**'s RTOS and TCP/IP awareness

µC/Probe, Graphical Live Watch®



- (1) Load the **.ELF** from the build
  - You have access to ***all global variables*** by their name
- (2) Drag-and-drop graphical objects from the palette
- (3) Assign variables (by name) to:
  - Gauges, meters, bar graphs, cylinders, etc.
  - Numeric indicators, sliders, switches, etc.
  - Built-in oscilloscope (up to 8 channels)
  - Excel spreadsheet interface
  - Scripting
  - Terminal window
- (4) Run – starts collecting the current value of the selected variables.
  - Don't have to stop the target!

µC/Probe, Graphical Live Watch® - Advanced Features

Micrium's µC/Probe™

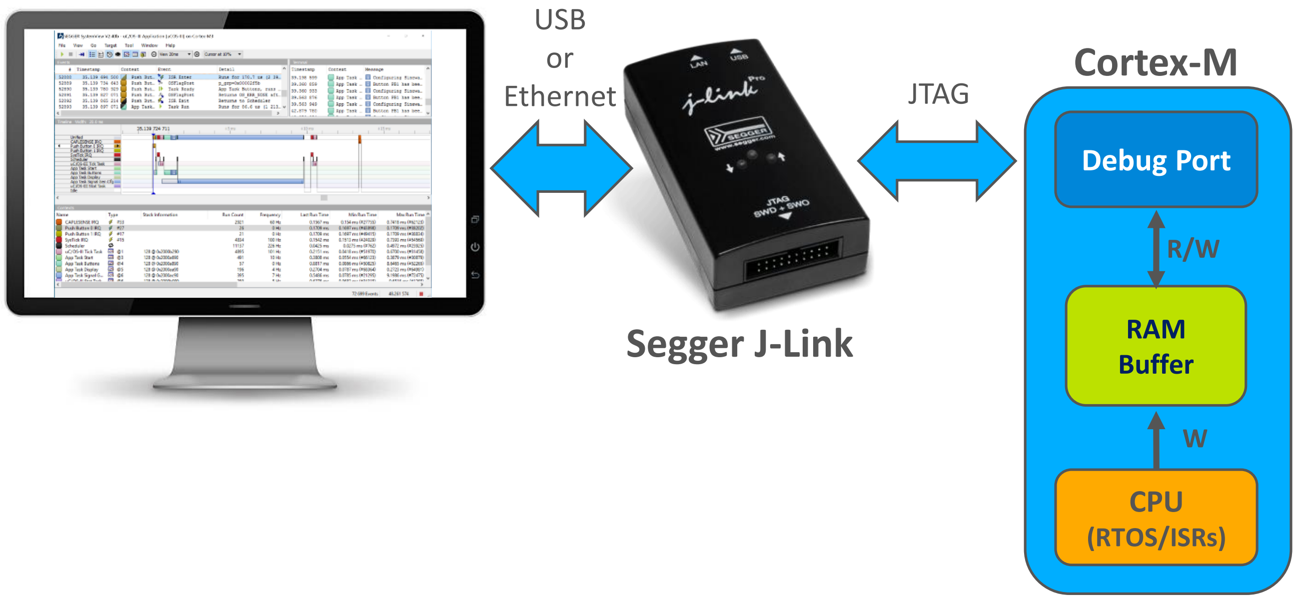
- 8-channel oscilloscope
  - No need to instrument your code and bring out signals
- Charts (trends)
- Excel spreadsheet interface
- Terminal window
- RTOS awareness
  - CPU usage of a per-task basis
  - ISR and task stack usage on a per-task basis
  - Status of all kernel objects
- TCP/IP Awareness
  - Buffer usage
  - Interface status (Ethernet or Wi-Fi)
  - Data transfer rates
- More

µC/Probe DEMO

Segger’s SystemView

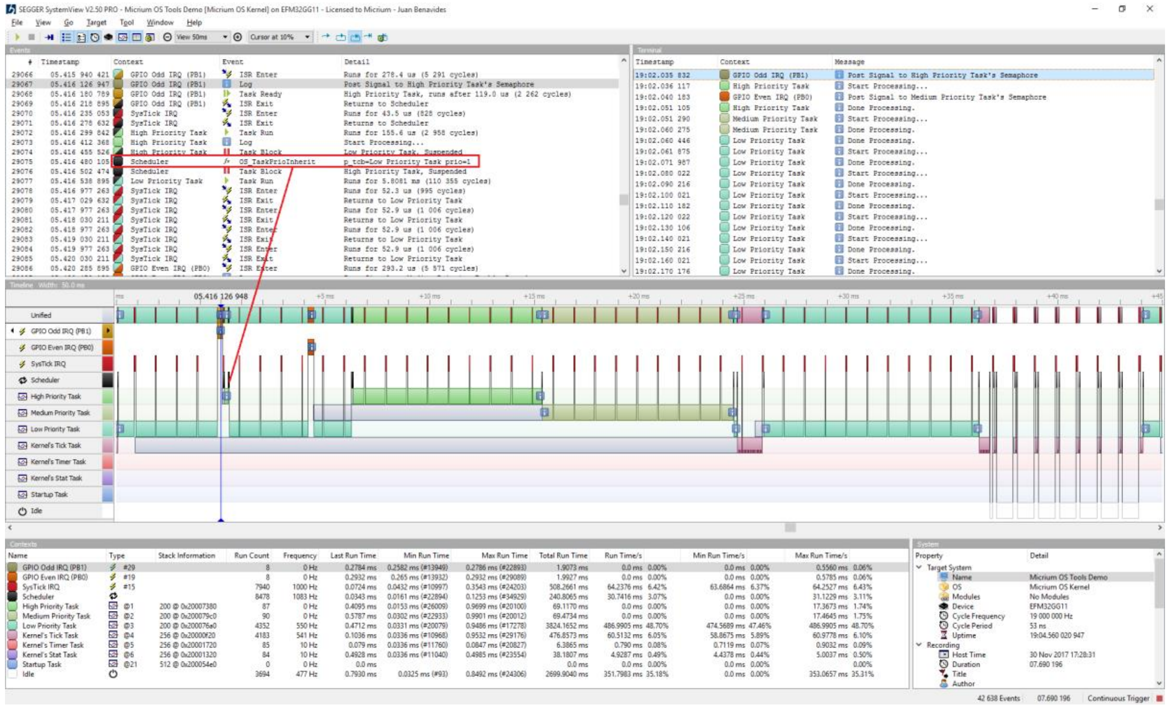
(www.segger.com)

Segger's SystemView



- Typically used in an RTOS-based system
  - The RTOS needs to be '**instrumented**'
  - Supports:
    - µC/OS-III,
    - Micrium OS Kernel,
    - embOS and
    - FreeRTOS
- Events are 'recorded' into a RAM buffer
  - ISR enter/exit
  - Semaphore pend/post
  - Mutex pend/post
  - Message queue pend/post
  - User Events
  - Etc.

Debugging RTOS-Based Systems – Segger's SystemView



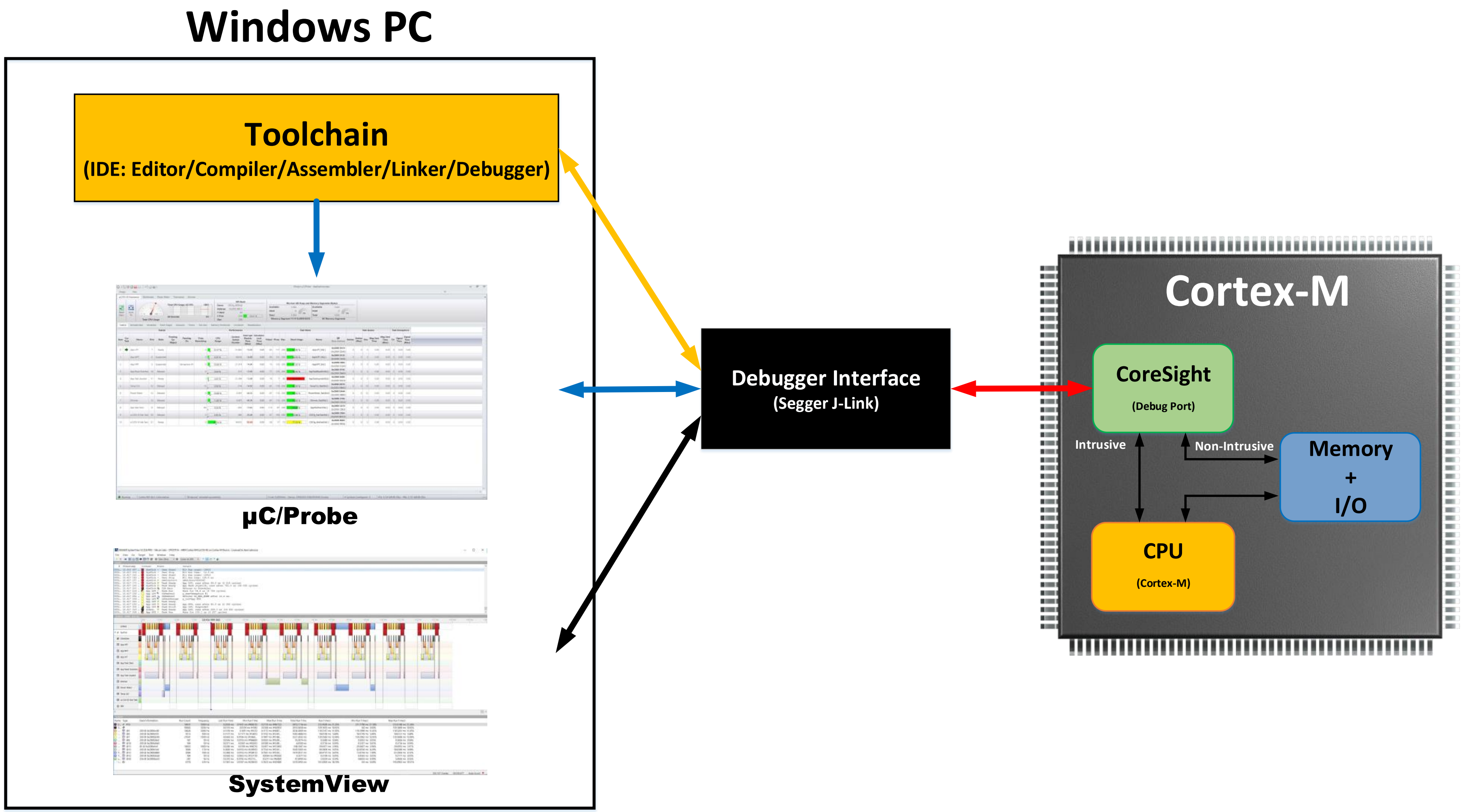
- Displays the execution profile of RTOS-based systems
  - Displayed **live**
    - Trigger on any task or ISR
  - Visualizing the execution profile of an application
  - Helps confirm the expected behavior of your system
- Measures CPU usage on a per-task basis
  - Min/Max/Avg task run time
  - Counts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

- www.Segger.com

SystemView Demo

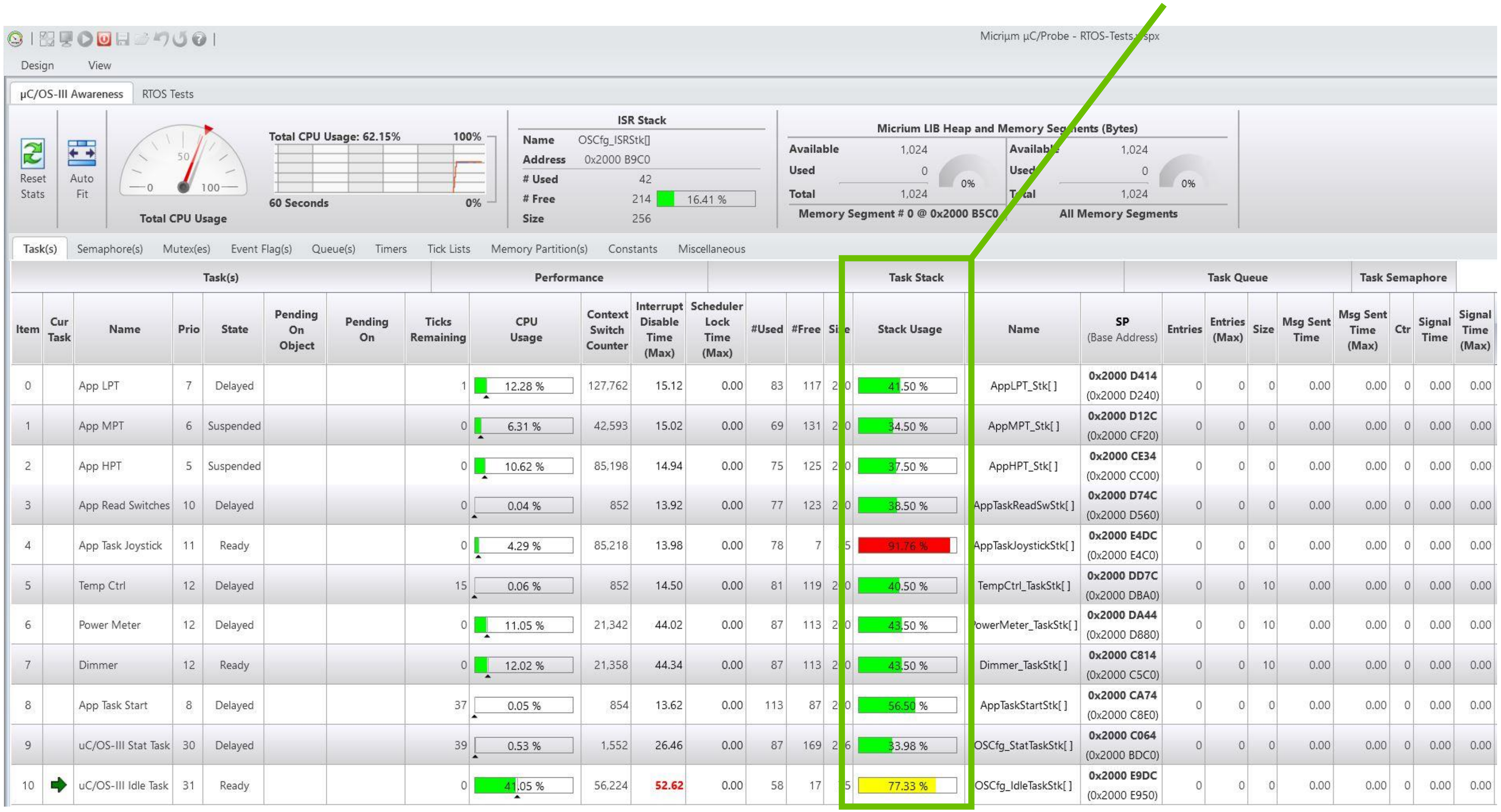
Debugging RTOS-based Systems

Tools for Testing/Debugging RTOS-based Systems



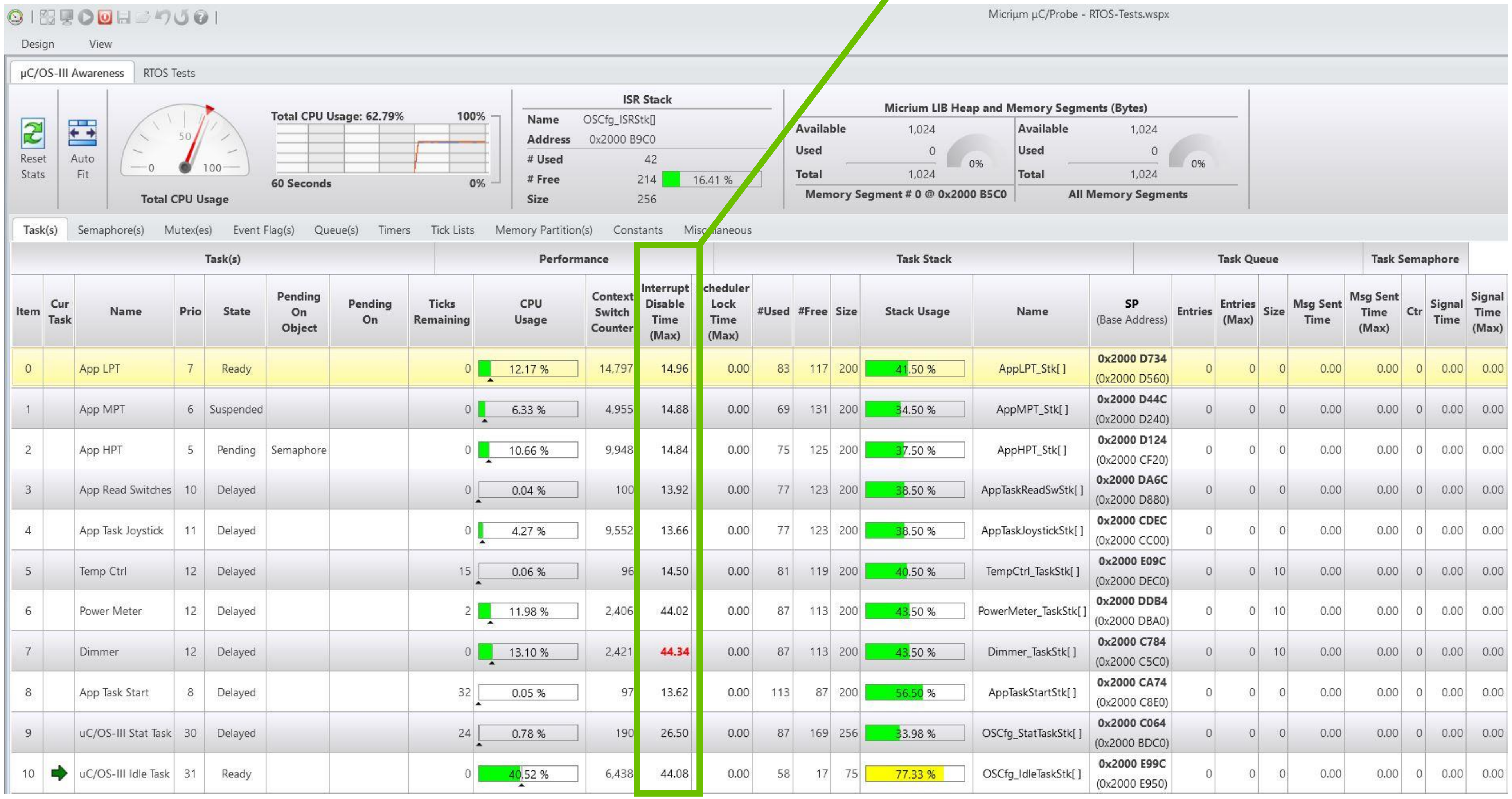
Detecting Stack Overflows – Detected with µC/Probe

Red shows stack close to overflowing



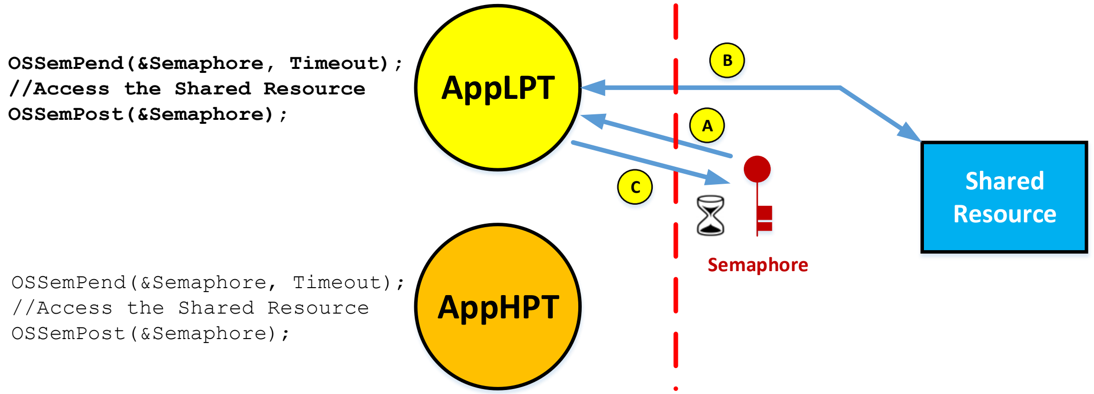
Interrupt Disable Time – Detected with µC/Probe

Long interrupt disable time affects system responsiveness

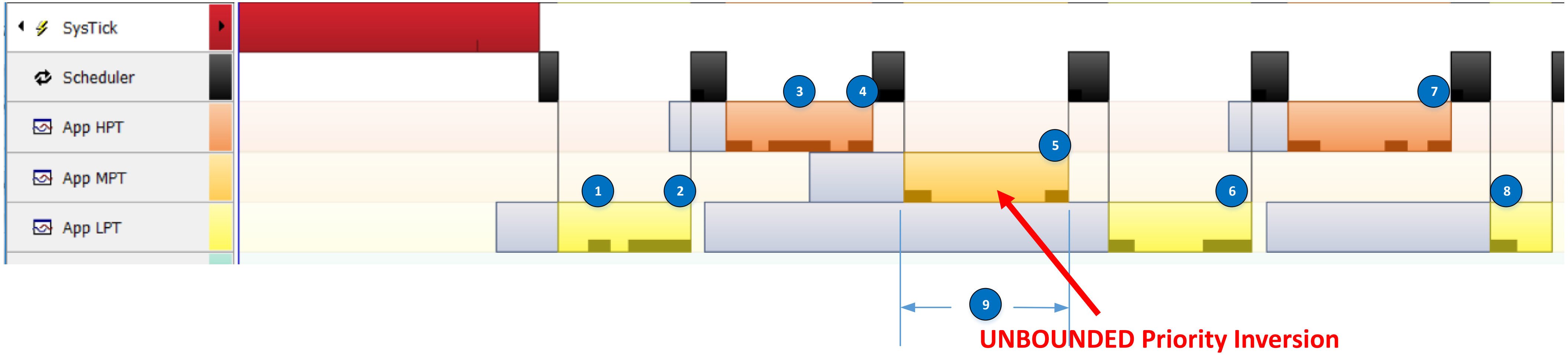


Task(s)								Performance				Task Stack						Task Queue					Task Semaphore		
Item	Cur Task	Name	Prio	State	Pending On Object	Pending On	Ticks Remaining	CPU Usage	Context Switch Counter	Interrupt Disable Time (Max)	cheduler Lock Time (Max)	#Used	#Free	Size	Stack Usage	Name	SP (Base Address)	Entries	Entries (Max)	Size	Msg Sent Time	Msg Sent Time (Max)	Ctr	Signal Time	Signal Time (Max)
0		App LPT	7	Ready			0	12.17 %	14,797	14.96	0.00	83	117	200	41.50 %	AppLPT_Stk[ ]	0x2000 D734 (0x2000 D560)	0	0	0	0.00	0.00	0	0.00	0.00
1		App MPT	6	Suspended			0	6.33 %	4,955	14.88	0.00	69	131	200	34.50 %	AppMPT_Stk[ ]	0x2000 D44C (0x2000 D240)	0	0	0	0.00	0.00	0	0.00	0.00
2		App HPT	5	Pending	Semaphore		0	10.66 %	9,948	14.84	0.00	75	125	200	37.50 %	AppHPT_Stk[ ]	0x2000 D124 (0x2000 CF20)	0	0	0	0.00	0.00	0	0.00	0.00
3		App Read Switches	10	Delayed			0	0.04 %	100	13.92	0.00	77	123	200	38.50 %	AppTaskReadSwStk[ ]	0x2000 DA6C (0x2000 D880)	0	0	0	0.00	0.00	0	0.00	0.00
4		App Task Joystick	11	Delayed			0	4.27 %	9,552	13.66	0.00	77	123	200	38.50 %	AppTaskJoystickStk[ ]	0x2000 CDEC (0x2000 CC00)	0	0	0	0.00	0.00	0	0.00	0.00
5		Temp Ctrl	12	Delayed			15	0.06 %	96	14.50	0.00	81	119	200	40.50 %	TempCtrl_TaskStk[ ]	0x2000 E09C (0x2000 DEC0)	0	0	10	0.00	0.00	0	0.00	0.00
6		Power Meter	12	Delayed			2	11.98 %	2,406	44.02	0.00	87	113	200	43.50 %	PowerMeter_TaskStk[ ]	0x2000 DDB4 (0x2000 DBA0)	0	0	10	0.00	0.00	0	0.00	0.00
7		Dimmer	12	Delayed			0	13.10 %	2,421	**44.34**	0.00	87	113	200	43.50 %	Dimmer_TaskStk[ ]	0x2000 C784 (0x2000 C5C0)	0	0	10	0.00	0.00	0	0.00	0.00
8		App Task Start	8	Delayed			32	0.05 %	97	13.62	0.00	113	87	200	56.50 %	AppTaskStartStk[ ]	0x2000 CA74 (0x2000 C8E0)	0	0	0	0.00	0.00	0	0.00	0.00
9		uC/OS-III Stat Task	30	Delayed			24	0.78 %	190	26.50	0.00	87	169	256	33.98 %	OSCfg_StatTaskStk[ ]	0x2000 C064 (0x2000 BDC0)	0	0	0	0.00	0.00	0	0.00	0.00
10	➡	uC/OS-III Idle Task	31	Ready			0	40.52 %	6,438	44.08	0.00	58	17	75	77.33 %	OSCfg_IdleTaskStk[ ]	0x2000 E99C (0x2000 E950)	0	0	0	0.00	0.00	0	0.00	0.00

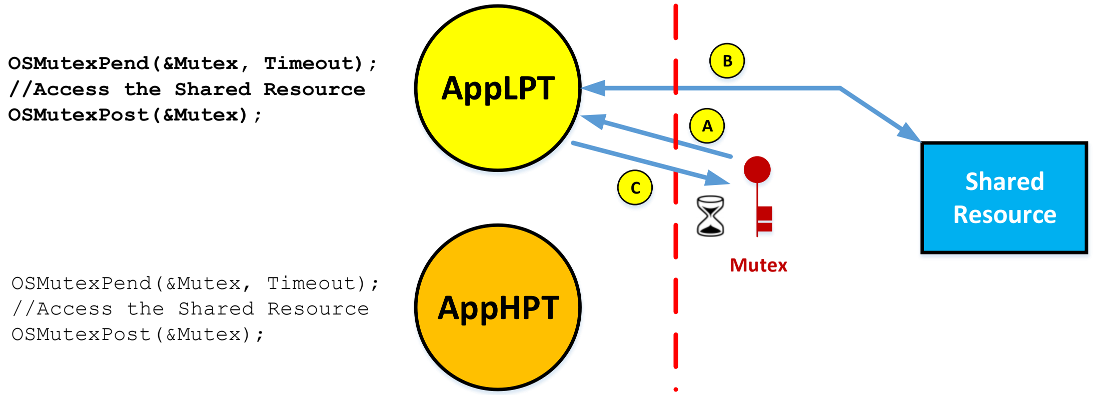
Priority Inversions Problem – Detected with SystemView



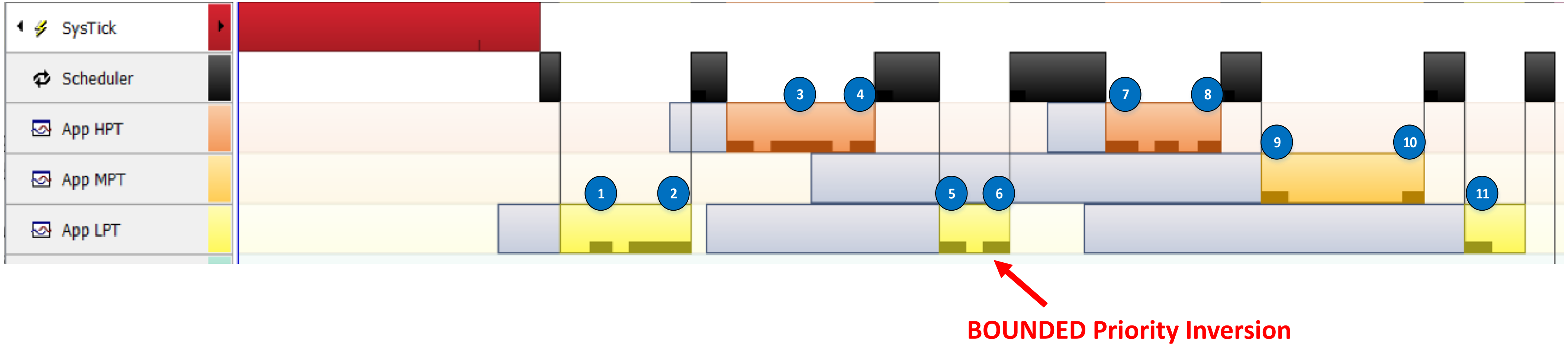
Priority Inversion caused by using a Semaphore:



Priority Inversion Solution – Confirmed with SystemView

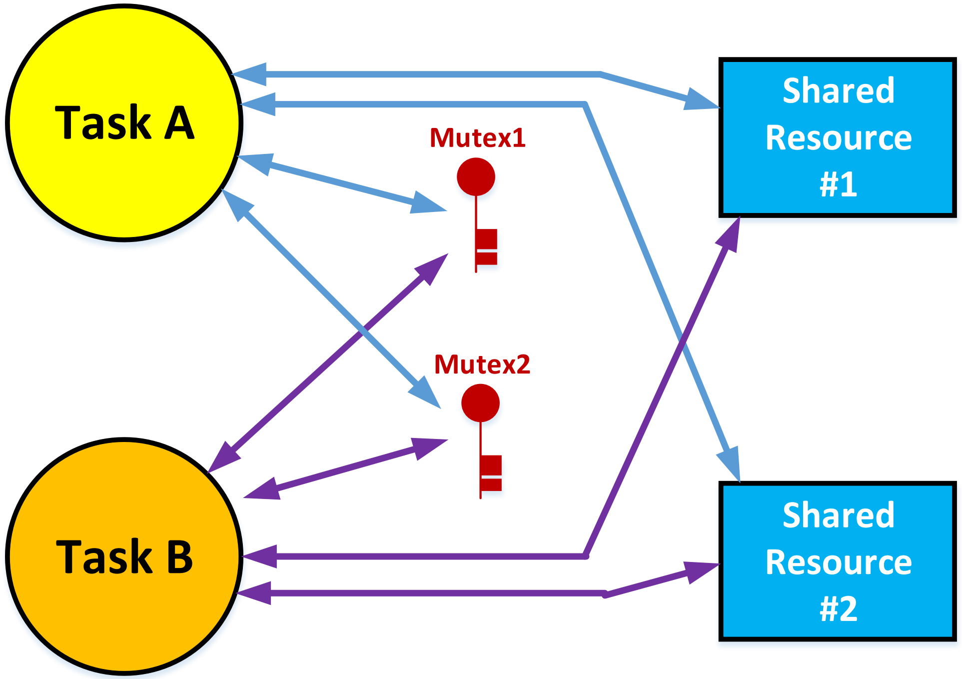


Unbounded Priority Inversion eliminated by using a Mutex



Deadlock Problem

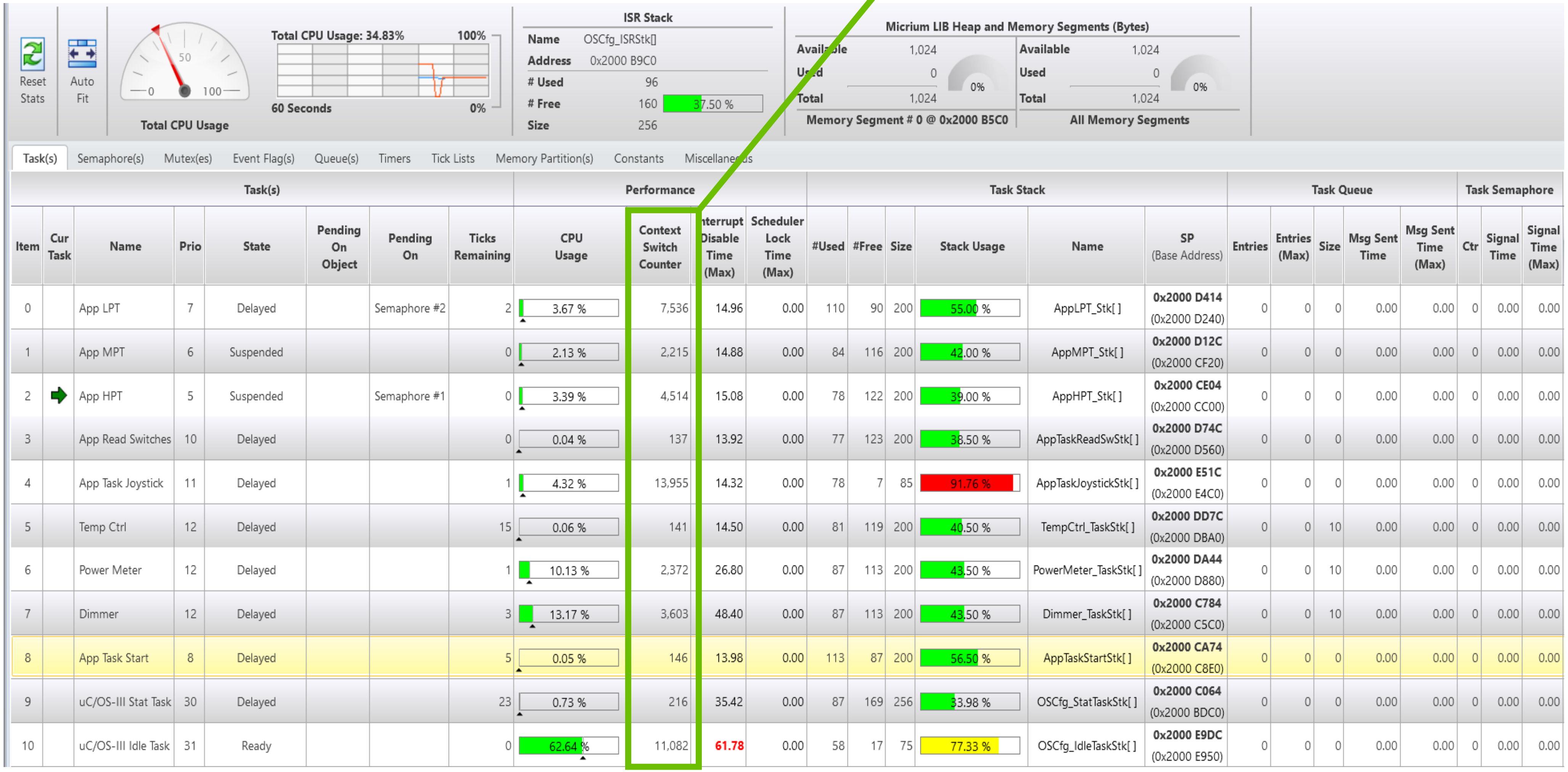
```
OSMutexPend(&Mutex1, Timeout);
OSMutexPend(&Mutex2, Timeout);
//Access the Shared Resource
OSMutexPost(&Mutex2);
OSMutexPost(&Mutex1);
```




```
OSMutexPend(&Mutex2, Timeout);
OSMutexPend(&Mutex1, Timeout);
//Access the Shared Resource
OSMutexPost(&Mutex1);
OSMutexPost(&Mutex2);
```

Deadlocks - Detected with µC/Probe

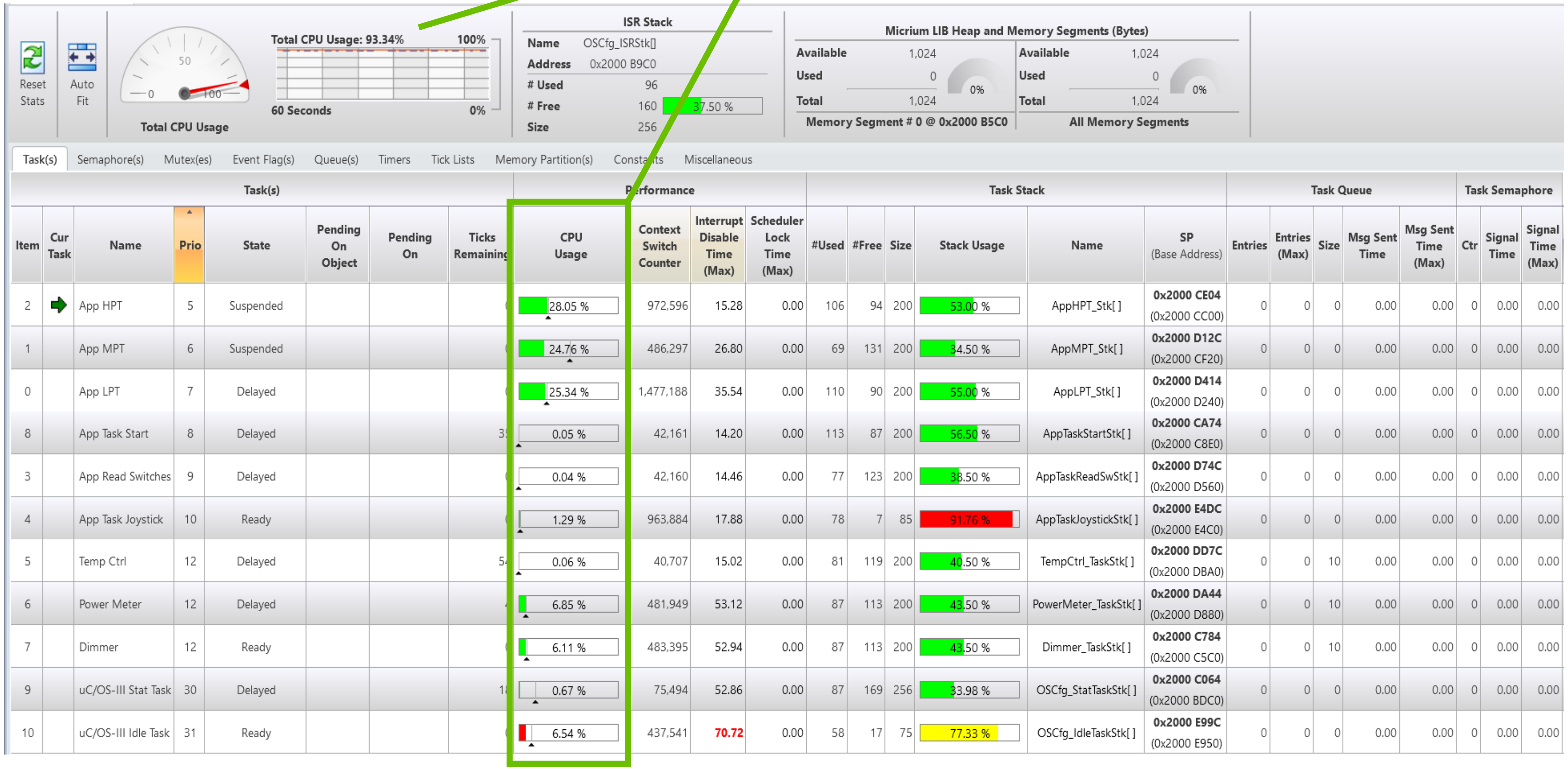
Two or more tasks would stop executing



Task(s)								Performance				Task Stack						Task Queue					Task Semaphore		
Item	Cur Task	Name	Prio	State	Pending On Object	Pending On	Ticks Remaining	CPU Usage	Context Switch Counter	Interrupt Disable Time (Max)	Scheduler Lock Time (Max)	#Used	#Free	Size	Stack Usage	Name	SP (Base Address)	Entries	Entries (Max)	Size	Msg Sent Time	Msg Sent Time (Max)	Ctr	Signal Time	Signal Time (Max)
0		App LPT	7	Delayed		Semaphore #2	2	3.67 %	7,536	14.96	0.00	110	90	200	55.00 %	AppLPT_Stk[ ]	0x2000 D414 (0x2000 D240)	0	0	0	0.00	0.00	0	0.00	0.00
1		App MPT	6	Suspended			0	2.13 %	2,215	14.88	0.00	84	116	200	42.00 %	AppMPT_Stk[ ]	0x2000 D12C (0x2000 CF20)	0	0	0	0.00	0.00	0	0.00	0.00
2	➡	App HPT	5	Suspended		Semaphore #1	0	3.39 %	4,514	15.08	0.00	78	122	200	39.00 %	AppHPT_Stk[ ]	0x2000 CE04 (0x2000 CC00)	0	0	0	0.00	0.00	0	0.00	0.00
3		App Read Switches	10	Delayed			0	0.04 %	137	13.92	0.00	77	123	200	38.50 %	AppTaskReadSwStk[ ]	0x2000 D74C (0x2000 D560)	0	0	0	0.00	0.00	0	0.00	0.00
4		App Task Joystick	11	Delayed			1	4.32 %	13,955	14.32	0.00	78	7	85	91.76 %	AppTaskJoystickStk[ ]	0x2000 E51C (0x2000 E4C0)	0	0	0	0.00	0.00	0	0.00	0.00
5		Temp Ctrl	12	Delayed			15	0.06 %	141	14.50	0.00	81	119	200	40.50 %	TempCtrl_TaskStk[ ]	0x2000 DD7C (0x2000 DBA0)	0	0	10	0.00	0.00	0	0.00	0.00
6		Power Meter	12	Delayed			1	10.13 %	2,372	26.80	0.00	87	113	200	43.50 %	PowerMeter_TaskStk[ ]	0x2000 DA44 (0x2000 D880)	0	0	10	0.00	0.00	0	0.00	0.00
7		Dimmer	12	Delayed			3	13.17 %	3,603	48.40	0.00	87	113	200	43.50 %	Dimmer_TaskStk[ ]	0x2000 C784 (0x2000 C5C0)	0	0	10	0.00	0.00	0	0.00	0.00
8		App Task Start	8	Delayed			5	0.05 %	146	13.98	0.00	113	87	200	56.50 %	AppTaskStartStk[ ]	0x2000 CA74 (0x2000 C8E0)	0	0	0	0.00	0.00	0	0.00	0.00
9		uC/OS-III Stat Task	30	Delayed			23	0.73 %	216	35.42	0.00	87	169	256	33.98 %	OSCfg_StatTaskStk[ ]	0x2000 C064 (0x2000 BDC0)	0	0	0	0.00	0.00	0	0.00	0.00
10		uC/OS-III Idle Task	31	Ready			0	62.64 %	11,082	61.78	0.00	58	17	75	77.33 %	OSCfg_IdleTaskStk[ ]	0x2000 E9DC (0x2000 E950)	0	0	0	0.00	0.00	0	0.00	0.00

Starvation - Detected with µC/Probe

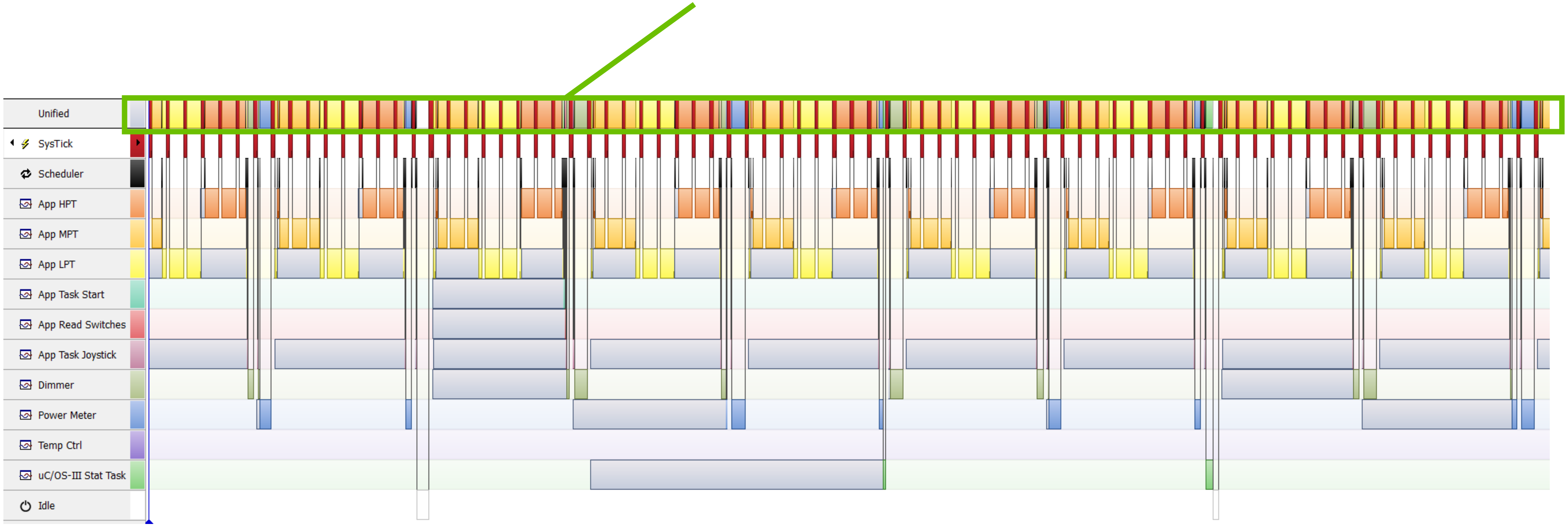
High CPU usage for high-priority task(s) can starve low-priority tasks



Task(s)								Performance				Task Stack						Task Queue					Task Semaphore		
Item	Cur Task	Name	Prio	State	Pending On Object	Pending On	Ticks Remaining	CPU Usage	Context Switch Counter	Interrupt Disable Time (Max)	Scheduler Lock Time (Max)	#Used	#Free	Size	Stack Usage	Name	SP (Base Address)	Entries	Entries (Max)	Size	Msg Sent Time	Msg Sent Time (Max)	Ctr	Signal Time	Signal Time (Max)
2	➡	App HPT	5	Suspended			0	28.05 %	972,596	15.28	0.00	106	94	200	53.00 %	AppHPT_Stk[ ]	0x2000 CE04 (0x2000 CC00)	0	0	0	0.00	0.00	0	0.00	0.00
1		App MPT	6	Suspended			0	24.76 %	486,297	26.80	0.00	69	131	200	34.50 %	AppMPT_Stk[ ]	0x2000 D12C (0x2000 CF20)	0	0	0	0.00	0.00	0	0.00	0.00
0		App LPT	7	Delayed			0	25.34 %	1,477,188	35.54	0.00	110	90	200	55.00 %	AppLPT_Stk[ ]	0x2000 D414 (0x2000 D240)	0	0	0	0.00	0.00	0	0.00	0.00
8		App Task Start	8	Delayed			35	0.05 %	42,161	14.20	0.00	113	87	200	56.50 %	AppTaskStartStk[ ]	0x2000 CA74 (0x2000 C8E0)	0	0	0	0.00	0.00	0	0.00	0.00
3		App Read Switches	9	Delayed			0	0.04 %	42,160	14.46	0.00	77	123	200	38.50 %	AppTaskReadSwStk[ ]	0x2000 D74C (0x2000 D560)	0	0	0	0.00	0.00	0	0.00	0.00
4		App Task Joystick	10	Ready			0	1.29 %	963,884	17.88	0.00	78	7	85	91.76 %	AppTaskJoystickStk[ ]	0x2000 E4DC (0x2000 E4C0)	0	0	0	0.00	0.00	0	0.00	0.00
5		Temp Ctrl	12	Delayed			54	0.06 %	40,707	15.02	0.00	81	119	200	40.50 %	TempCtrl_TaskStk[ ]	0x2000 DD7C (0x2000 DBA0)	0	0	10	0.00	0.00	0	0.00	0.00
6		Power Meter	12	Delayed			4	6.85 %	481,949	53.12	0.00	87	113	200	43.50 %	PowerMeter_TaskStk[ ]	0x2000 DA44 (0x2000 D880)	0	0	10	0.00	0.00	0	0.00	0.00
7		Dimmer	12	Ready			0	6.11 %	483,395	52.94	0.00	87	113	200	43.50 %	Dimmer_TaskStk[ ]	0x2000 C784 (0x2000 C5C0)	0	0	10	0.00	0.00	0	0.00	0.00
9		uC/OS-III Stat Task	30	Delayed			18	0.67 %	75,494	52.86	0.00	87	169	256	33.98 %	OSCfg_StatTaskStk[ ]	0x2000 C064 (0x2000 BDC0)	0	0	0	0.00	0.00	0	0.00	0.00
10		uC/OS-III Idle Task	31	Ready			0	6.54 %	437,541	70.72	0.00	58	17	75	77.33 %	OSCfg_IdleTaskStk[ ]	0x2000 E99C (0x2000 E950)	0	0	0	0.00	0.00	0	0.00	0.00

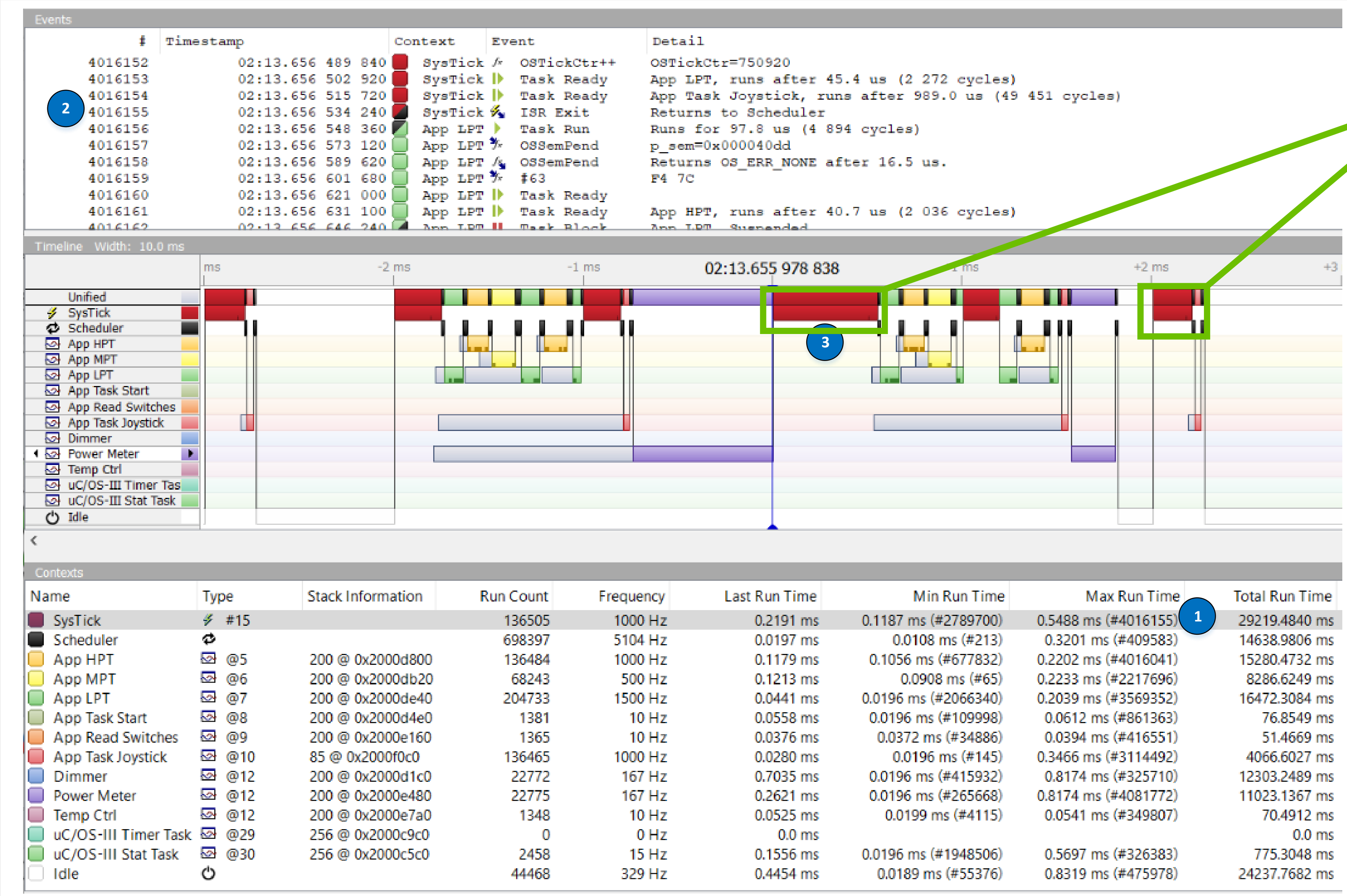
Starvation - Detected with SystemView

Excessive CPU Usage means that low-priority tasks are subject to starvation



Code Execution Time - Detected with SystemView

SysTick ISR execution time longer than usual!?!



#	Timestamp	Context	Event	Detail
4016152	02:13.656 489 840	SysTick	OSTickCtr++	OSTickCtr=750920
4016153	02:13.656 502 920	SysTick	Task Ready	App LPT, runs after 45.4 us (2 272 cycles)
4016154	02:13.656 515 720	SysTick	Task Ready	App Task Joystick, runs after 989.0 us (49 451 cycles)
4016155	02:13.656 534 240	SysTick	ISR Exit	Returns to Scheduler
4016156	02:13.656 548 360	App LPT	Task Run	Runs for 97.8 us (4 894 cycles)
4016157	02:13.656 573 120	App LPT	OSSemPend	p_sem=0x000040dd
4016158	02:13.656 589 620	App LPT	OSSemPend	Returns OS_ERR_NONE after 16.5 us.
4016159	02:13.656 601 680	App LPT	#63	F4 7C
4016160	02:13.656 621 000	App LPT	Task Ready	
4016161	02:13.656 631 100	App LPT	Task Ready	App HPT, runs after 40.7 us (2 036 cycles)
4016162	02:13.656 646 240	App LPT	Task Block	App LPT Suspended

Name	Type	Stack Information	Run Count	Frequency	Last Run Time	Min Run Time	Max Run Time	Total Run Time
SysTick	#15		136505	1000 Hz	0.2191 ms	0.1187 ms (#2789700)	0.5488 ms (#4016155)	29219.4840 ms
Scheduler			698397	5104 Hz	0.0197 ms	0.0108 ms (#213)	0.3201 ms (#409583)	14638.9806 ms
App HPT	@5	200 @ 0x2000d800	136484	1000 Hz	0.1179 ms	0.1056 ms (#677832)	0.2202 ms (#4016041)	15280.4732 ms
App MPT	@6	200 @ 0x2000db20	68243	500 Hz	0.1213 ms	0.0908 ms (#65)	0.2233 ms (#2217696)	8286.6249 ms
App LPT	@7	200 @ 0x2000de40	204733	1500 Hz	0.0441 ms	0.0196 ms (#2066340)	0.2039 ms (#3569352)	16472.3084 ms
App Task Start	@8	200 @ 0x2000d4e0	1381	10 Hz	0.0558 ms	0.0196 ms (#109998)	0.0612 ms (#861363)	76.8549 ms
App Read Switches	@9	200 @ 0x2000e160	1365	10 Hz	0.0376 ms	0.0372 ms (#34886)	0.0394 ms (#416551)	51.4669 ms
App Task Joystick	@10	85 @ 0x2000f0c0	136465	1000 Hz	0.0280 ms	0.0196 ms (#145)	0.3466 ms (#3114492)	4066.6027 ms
Dimmer	@12	200 @ 0x2000d1c0	22772	167 Hz	0.7035 ms	0.0196 ms (#415932)	0.8174 ms (#325710)	12303.2489 ms
Power Meter	@12	200 @ 0x2000e480	22775	167 Hz	0.2621 ms	0.0196 ms (#265668)	0.8174 ms (#4081772)	11023.1367 ms
Temp Ctrl	@12	200 @ 0x2000e7a0	1348	10 Hz	0.0525 ms	0.0199 ms (#4115)	0.0541 ms (#349807)	70.4912 ms
uC/OS-III Timer Task	@29	256 @ 0x2000c9c0	0	0 Hz	0.0 ms			0.0 ms
uC/OS-III Stat Task	@30	256 @ 0x2000c5c0	2458	15 Hz	0.1556 ms	0.0196 ms (#1948506)	0.5697 ms (#326383)	775.3048 ms
Idle			44468	329 Hz	0.4454 ms	0.0189 ms (#55376)	0.8319 ms (#475978)	24237.7682 ms

Code Execution Time – Displayed with µC/Probe

Code instrumented with Elapsed Time measurement functions:

```
elapsed_time_start(n);
// Code to measure
elapsed_time_stop(n);
```

```
void  elapsed_time_start (uint32_t  i)
{
    elapsed_time_tbl[i].start = ARM_CM_DWT_CYCCNT;
}
```

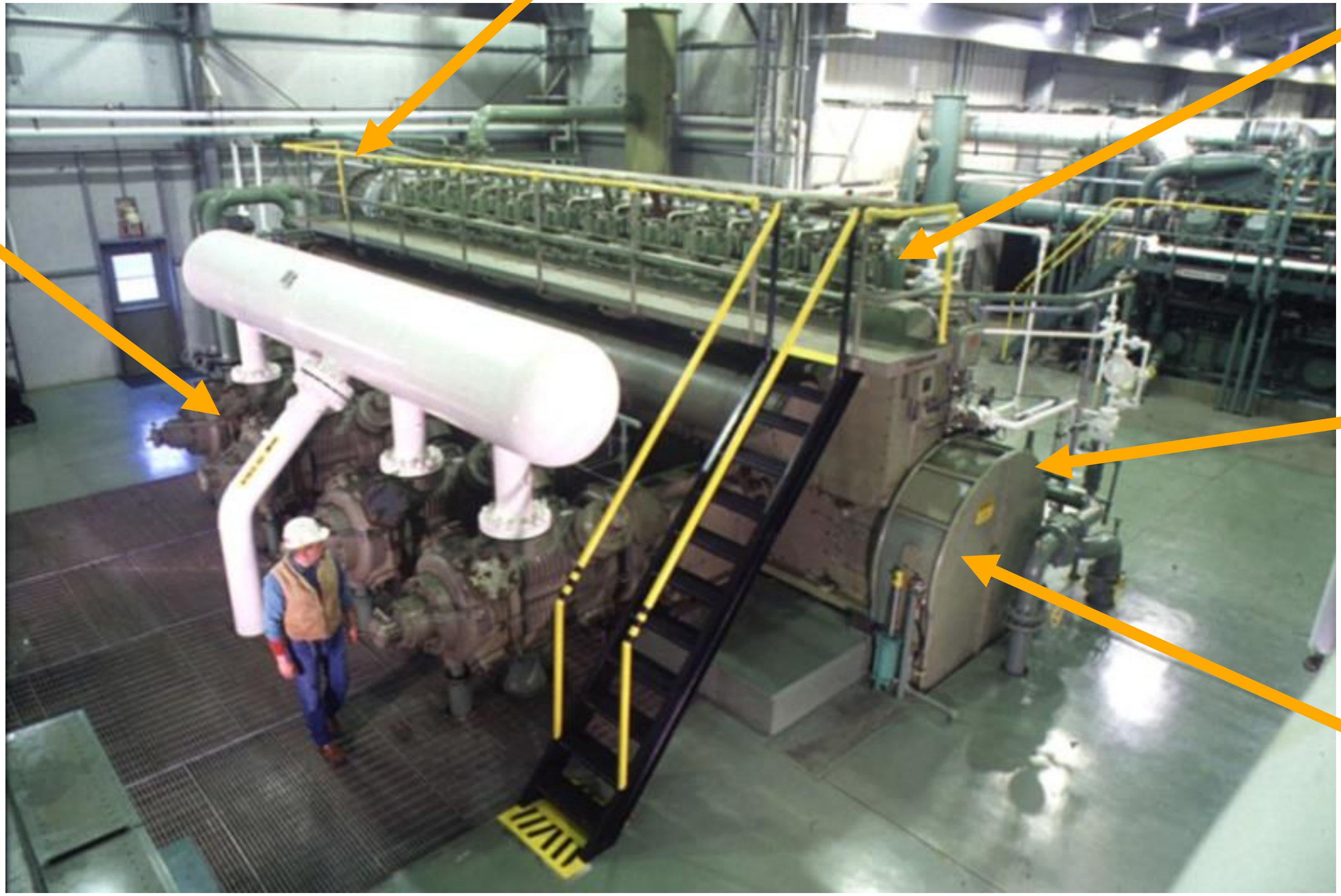
```
void  elapsed_time_stop (uint32_t  i)
{
    uint32_t       stop;
    ELAPSED_TIME  *p_tbl;


    stop           = ARM_CM_DWT_CYCCNT;
    p_tbl          = &elapsed_time_tbl[i];
    p_tbl->current = stop - p_tbl->start;
    if (p_tbl->max < p_tbl->current) {
        p_tbl->max = p_tbl->current;
    }
    if (p_tbl->min > p_tbl->current) {
        p_tbl->min = p_tbl->current;
    }
}
```

	OSTaskCreate() (microseconds)	OSSemCreate() (microseconds)	OSMutexCreate() (microseconds)
Max:	186.8	18.6	16.5
Min:	81.4	17.7	16.3

Scaled from Free-Running counter counts to microseconds

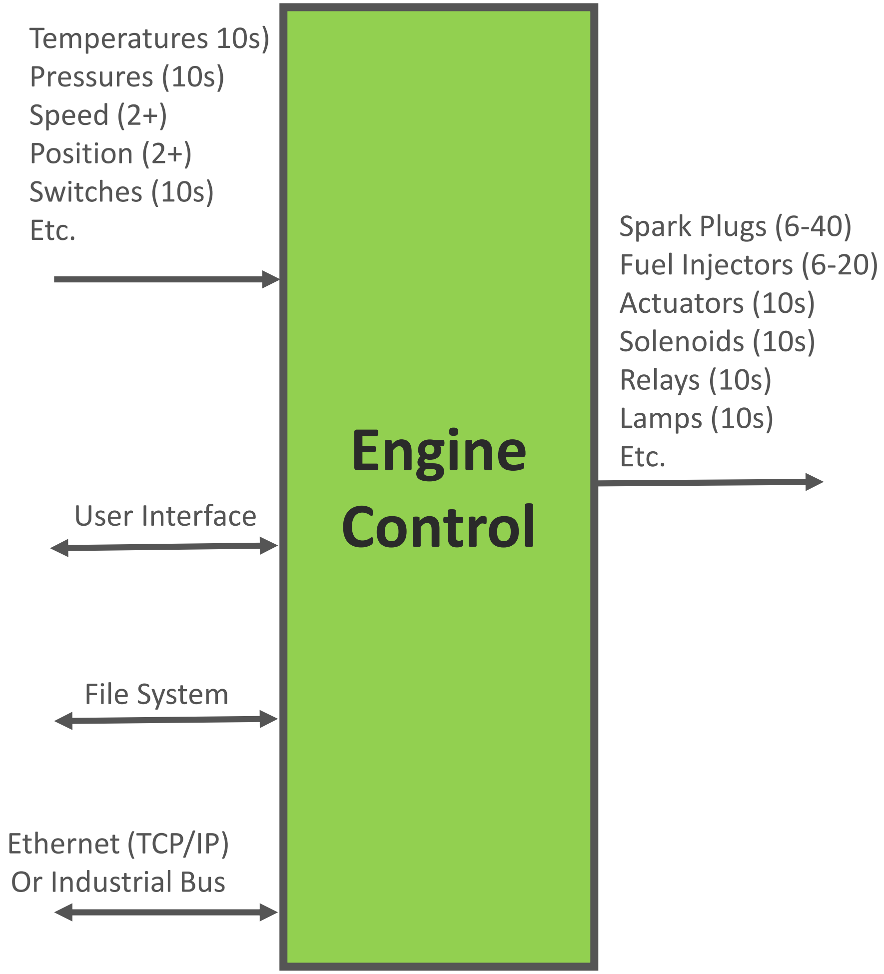
Using an RTOS – Industrial Engine Control

Natural Gas Compressor Stations (~300-600 RPM)

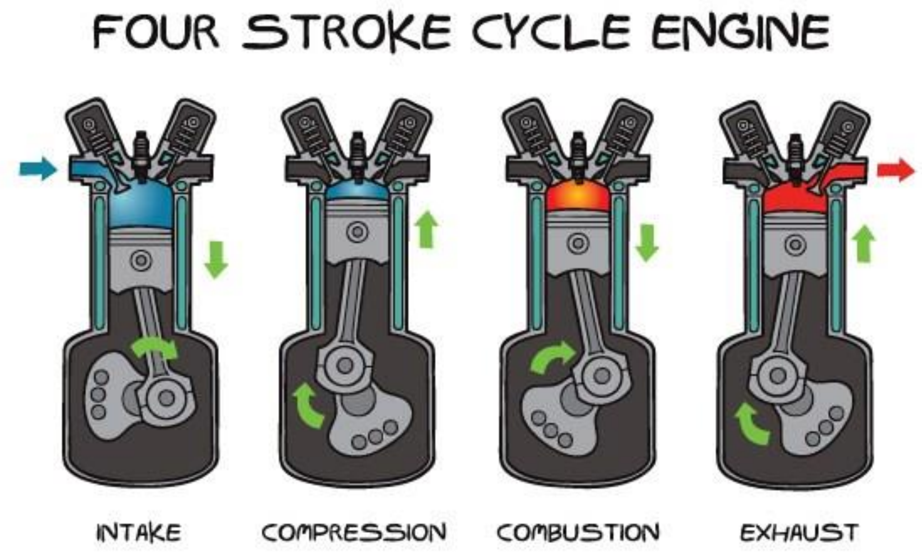


Using an RTOS – Industrial Engine Control

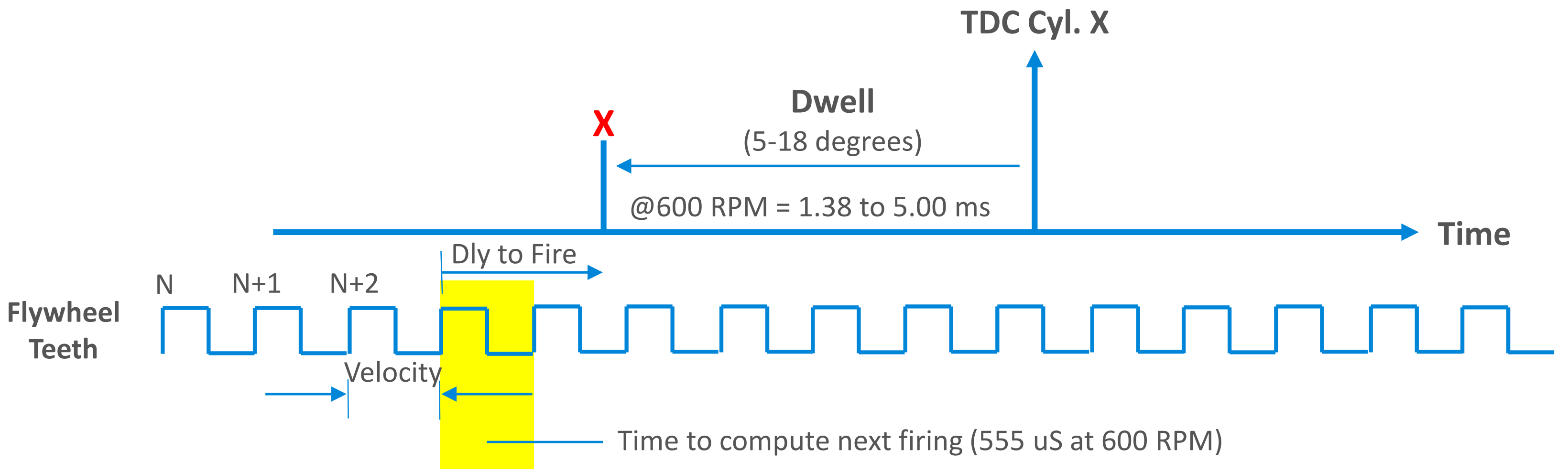
- Controls
  - Sequencing (Start, Load, Stop, Shutdown)
  - **Ignition (Time Critical)**
  - Fuel Management
    - **Fuel Injection (Time Critical)**
  - Air Management
    - Turbo charged
  - Valve Management
    - Suction, Discharge, Bypass
  - Compressor control
    - Loading with pockets (Open/Close, up to 32)
  - Lubrication control
- Monitoring
  - Temperatures
  - Pressures
  - Flow
  - Etc.



Ignition – Time Critical



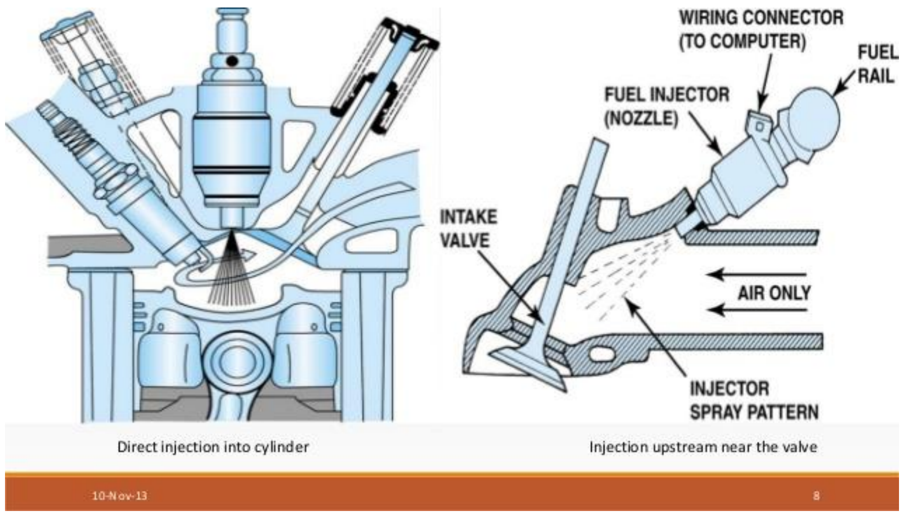
180 Teeth Ring Gear used for Timing



3 Sensors needed for timing:

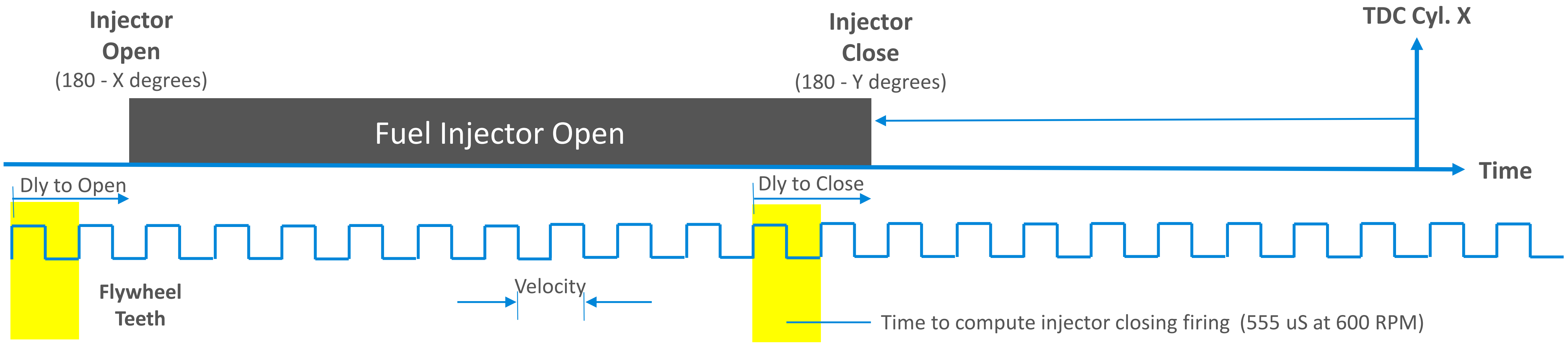
1) Flywheel position
2) TDC #1
3) CAM

Fuel Injection – Time Critical



3 Sensors needed for timing:

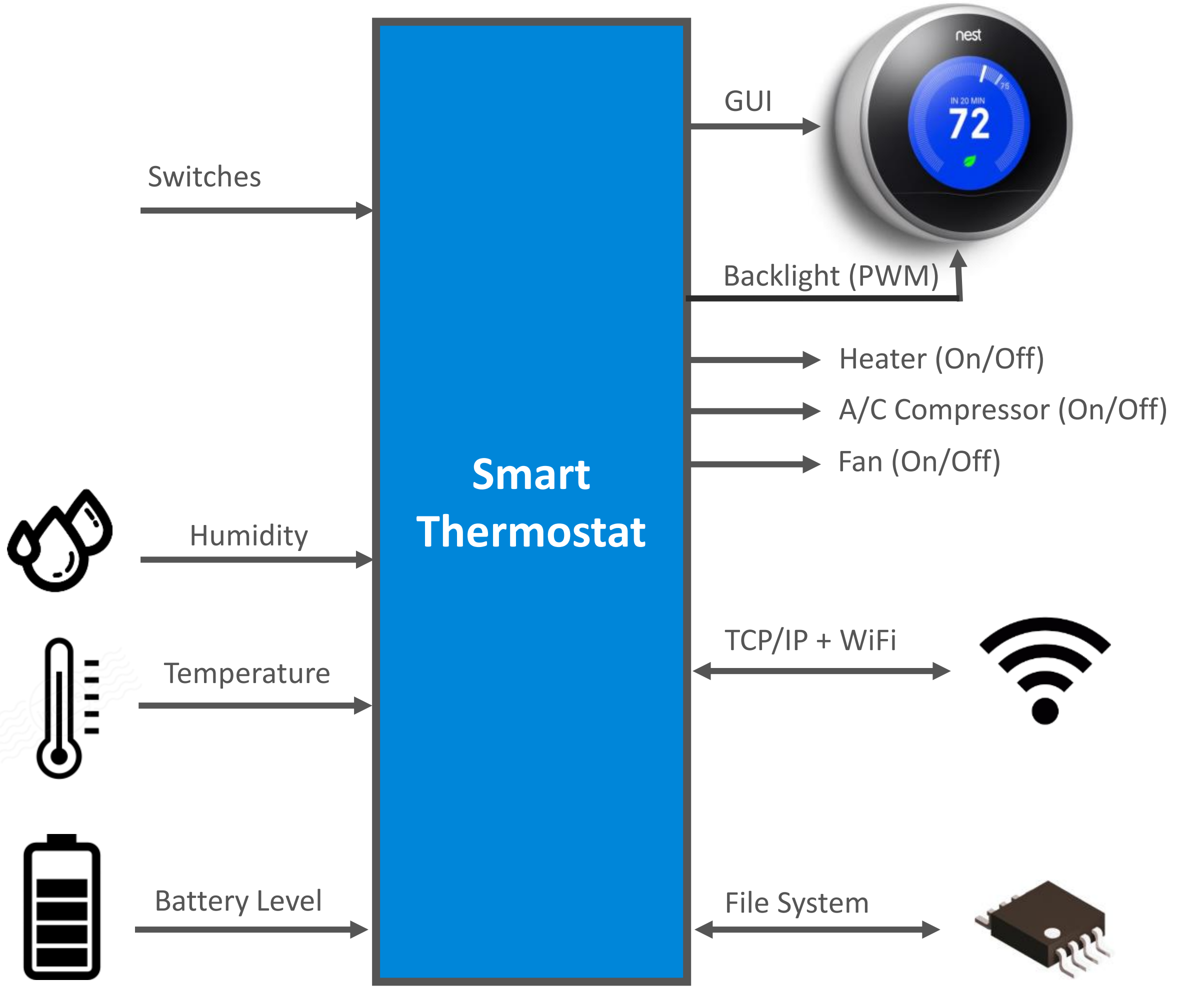
1) Flywheel position
2) TDC #1
3) CAM



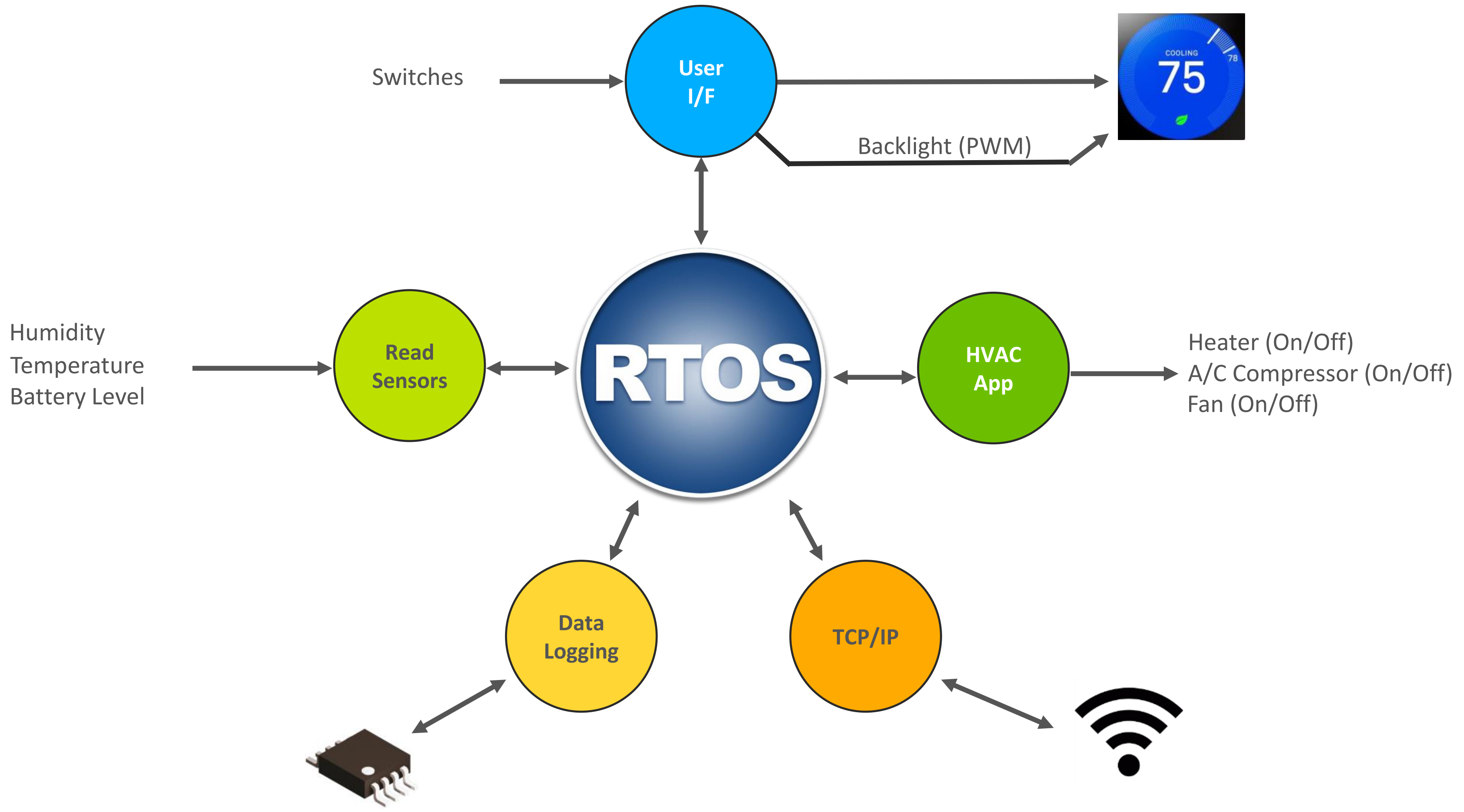
Using an RTOS – Smart Thermostat

Using an RTOS – An IoT Thermostat

- TCP/IP + WiFi
- Storage
- Rotary and push button interface
- Liquid Crystal Display (LCD)
- Backlight (brightness)
- Battery (monitoring)
- Sensors
  - Temperature
  - Humidity
  - Voltage
  - Presence
  - Etc.
- Controls
  - Heating Element
  - A/C Compressor
  - Fan



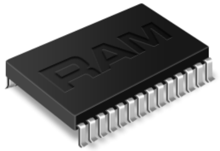
Using an RTOS – An IoT Thermostat – Task Diagram



Recommendations

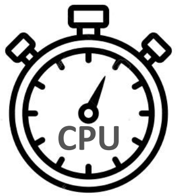
Recommendations - RTOS

- Don’t have too many tasks
  - Requires more RAM

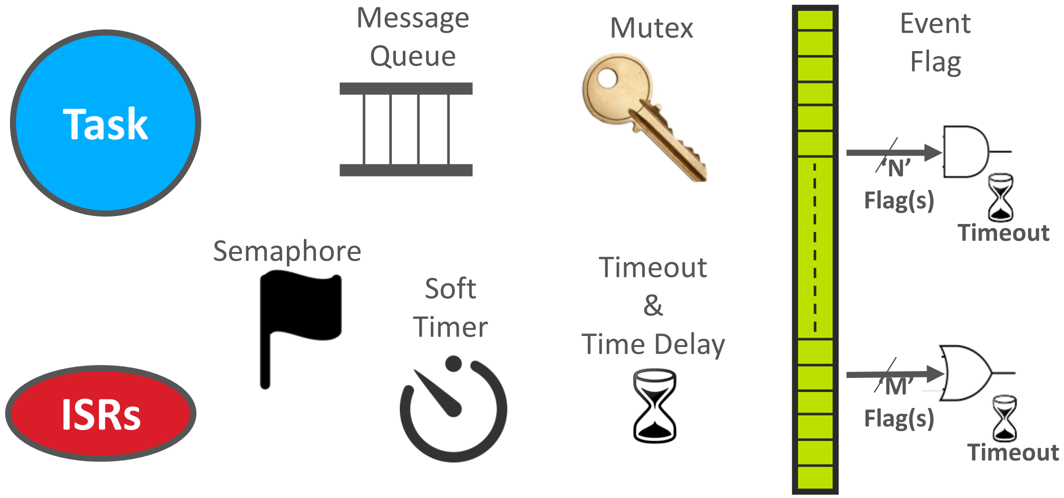


- Don’t have too few tasks
  - Defeats the purpose of having an RTOS
- Keep ISRs short
  - Clear the interrupt, signal a task
  - Use non-Kernel Aware ISRs only when absolutely needed
- Set task priorities at design time
  - Don’t change task priorities at run-time
- Use Mutexes instead of Semaphores for resource sharing
- Avoid using round-robin scheduling
  - Round-robin scheduling starve lower priority tasks

- Keep the number of priorities low (< 32)
  - More efficient scheduling
- RTOS APIs consume CPU cycles
  - Be aware of this



- Don’t enable the FPU if not needed
- Create graphical models of your application. Use:

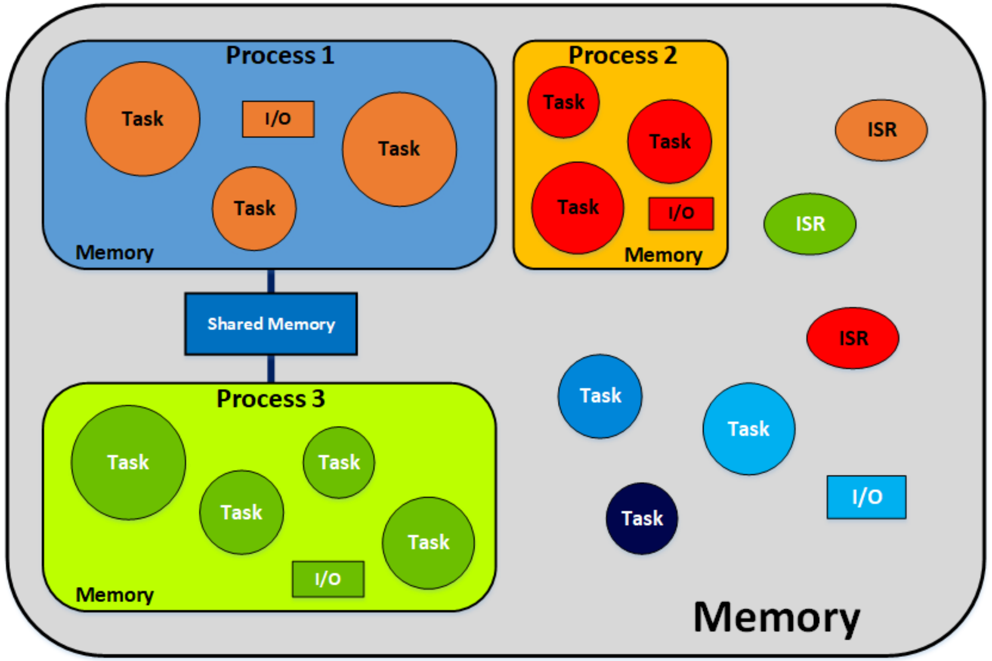


Recommendations - Storage

- Allocate all RTOS objects statically
  - Avoid `malloc()` and `free()`
- Don’t delete RTOS objects at run-time
  - If you `malloc()` don’t `free()`
  - The task could own resources that other tasks need
- Avoid excessive stack usage
  - Don’t allocate large arrays on task stacks
  - Some linkers will give you stack usage per function
  - Monitor stack usage using a Kernel aware debugger or µC/Probe
- Keep data in scope when using Message Queues

Recommendations – Use an MPU

- Separate the application by Process
  - Most tasks should be non-privileged
  - They cannot disable interrupts!
- Determine what to do when an access violation is detected
- Set the XN-bit (eXecute Never bit) for RAM
- Limit peripheral access to its own process
- Reduce interprocess communication
- Log/report faults to developers
- Create ‘named sections’ for your RAM
  - Makes it easier to map sections with the linker
- Don’t use a global heap
  - You cannot protect heap data with an MPU
- Don’t pass data from one task stack to another
- All kernel objects should be allocated in Kernel space
  - User task simply pass by reference



Recommendations – Use RTOS Aware Tools

- Use tools designed to debug RTOS-based applications

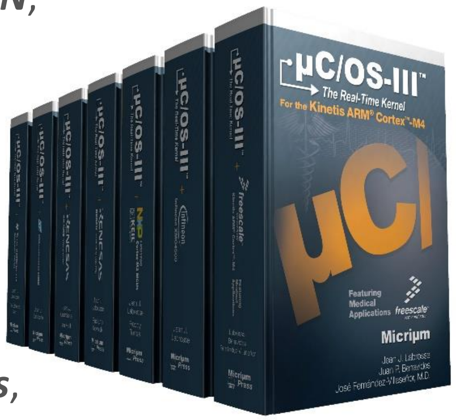
- Micrium's µC/Probe (www.micrium.com)
  - Provide 'visibility' in your running application
    - Any application variable can be displayed
  - Kernel Awareness
    - Monitor stack usage to detect potential overflows
    - Detect starvation
    - Detect deadlocks
    - Monitor CPU usage
    - Monitor interrupt disable time
    - Etc.
  - Simulate hardware
  - Change setpoints
  - Etc.

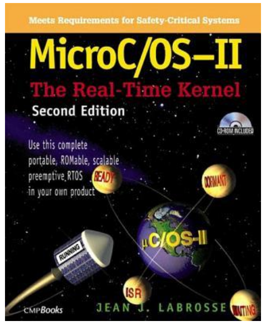
- Segger SystemView (www.segger.com)
  - Detect priority inversions
  - Detect starvation
  - Detect deadlocks
  - Measure code execution times
  - Validate priorities
  - Etc.

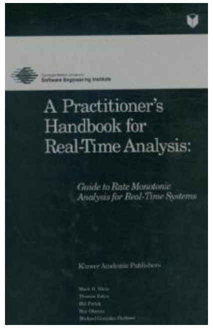
References

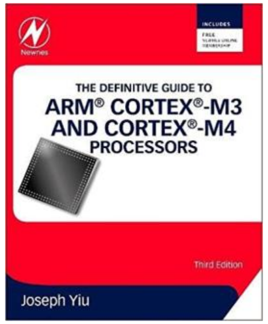
References – Books

- ***µC/OS-III, The Real-Time Kernel, and the Freescale Kinetis ARM Cortex-M4***, Jean J. Labrosse, 978-0982337523
- ***µC/OS-III, The Real-Time Kernel, and the Infineon XMC4500***, Jean J. Labrosse, 978-1935772200
- ***µC/OS-III, The Real-Time Kernel, and the NXP LPC1700***, Jean J. Labrosse, 978-0982337554
- ***µC/OS-III, The Real-Time Kernel, and the Renesas RX62N***, Jean J. Labrosse, 978-0982337578
- ***µC/OS-III, The Real-Time Kernel, and the Renesas SH7216***, Jean J. Labrosse, 978-0982337547
- ***µC/OS-III, The Real-Time Kernel, for the STM32 ARM Cortex-M3***, Jean J. Labrosse, 978-0982337530
- ***µC/OS-III, The Real-Time Kernel, and the Stellaris MCUs***, Jean J. Labrosse, 978-0982337561
- ***MicroC/OS-II, The Real-Time Kernel***, Jean J. Labrosse, 978-1578201037
- ***A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems***, by Mark Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael Gonzales Harbour, 978-0792393610
- ***The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors,*** Joseph Yiu, 978-0124080829



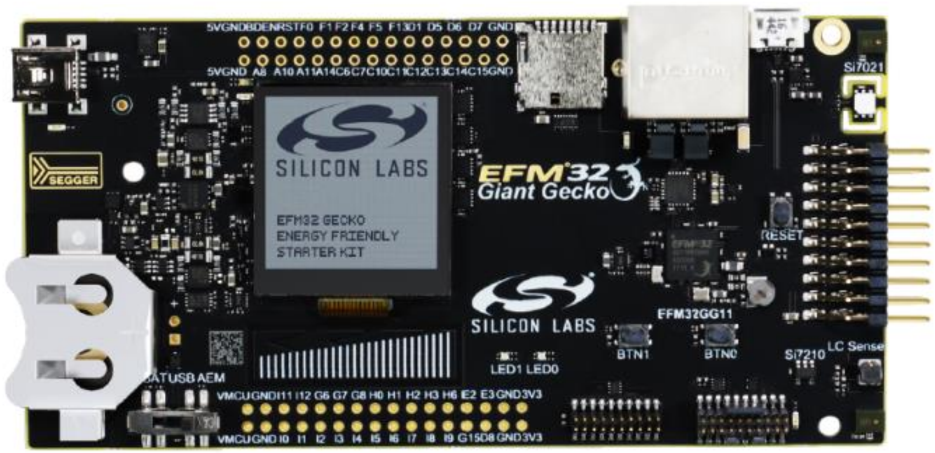






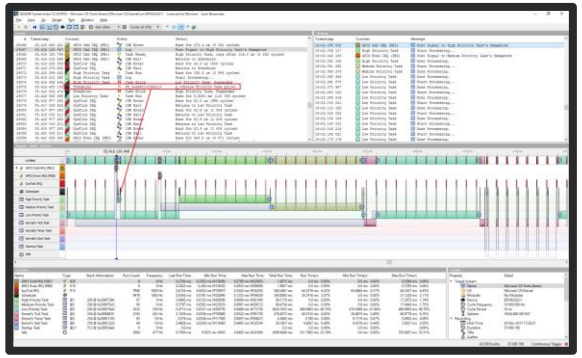
References - Development Tools

- 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 µC/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/



References – Videos

- ***Getting Started with Micrium OS, 10 Episode Series***
  - https://www.youtube.com/playlist?list=PL-awFRrdECXu9I7ybAl5tEgwn7BQF6N56
- ***SystemView for µC/OS-III***
  - https://www.youtube.com/watch?v=1Le5YwSADTs
- ***Micrium, Internet of Things***
  - https://www.micrium.com/training/videos/#foobox-3/0/SDJVFr4VUHA



References – Websites

- **Silicon Labs:**
  - Micrium OS Kernel (i.e. RTOS) **FREE** with Silicon Labs MCUs
  - Free development tools: Simplicity Studio
  - ***www.SiLabs.com***
- **Micrium (a Silicon Labs Business Unit):**
  - µC/OS-II and µC/OS-III RTOS and middleware
  - µC/Probe
  - Blogs
  - ***www.Micrium.com***
- **Segger:**
  - embOS RTOS and middleware
  - SystemView and J-Links
  - ***www.Segger.com***

Thank you!

SILABS.COM