ARM Cortex core microcontrollers

9. RTOS: Real-Time Operating Systems

Scherer Balázs

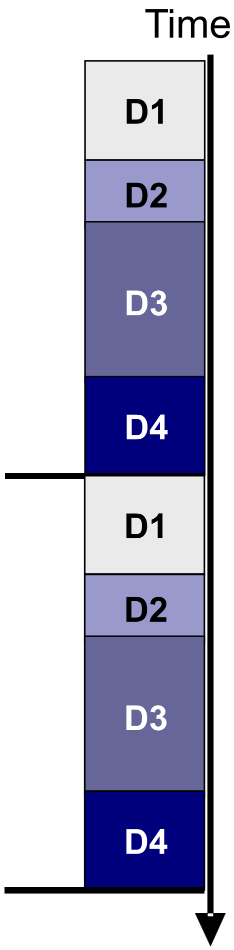
Méréstechnika és
Információs Rendszerek
Tanszék

MŰEGYETEM 1782

Embedded software architectures I.

- Round-Robin

```
void main(void)
{
  while(1)
  {
          if ( Device 1 needs service )
          {
            // Handle Device 1 and its data
          }
          if ( Device 2 needs service )
          {
            // Handle Device 2 and its data
          }
          if ( Device 3 needs service )
          {
            // Handle Device 3 and its data
          }
          ...
  }
}
```



Embedded software architectures II.

- Round-Robin
  - Very simple
  - No interrupt only main cycle
  - There is no shared resource problem

  - Worst case response time = Sum of the response times of the jobs
  - Worst Case increase linearly with the number of jobs
  - The Jitter is big
  - New jobs modifies the Jitter and the Worst case response time

Embedded software architectures III.

- Round-Robin with interrupts

```
BOOL Device1_flag = 0;
BOOL Device2_flag = 0;
BOOL Device3_flag = 0;

void interrupt vDevice1(void)
{
   // Handle Device 1 time critical part
   Device1_flag = 1;
}
void interrupt vDevice2(void)
{
   // Handle Device 2 time critical part
   Device1_flag = 2;
}
void interrupt vDevice3(void)
{
   // Handle Device 3 time critical part
   Device3_flag = 1;
}
```

```
void main(void)
{
  while(1)
  {
     if ( Device1_flag )
       {
          // Handle Device 1 and its data
       }
     if (Device2_flag )
       {
          // Handle Device 2 and its data
       }
     if (Device3_flag )
       {
          // Handle Device 3 and its data
       }
     ...
  }
}
```

Embedded software architectures IV.

- Round-Robin with interrupts
  - Better for handling time critical hardware
  - Shared resource problem present between the interrupt and the main cycle
  - Worst case response time = Sum of the response times of the jobs + IT
  - Worst Case increase linearly with the number of jobs
  - The Jitter is big
  - New jobs modifies the Jitter and the Worst case response time

Posibble problems I.: shared variables

- Not atomic variable handling can lead to problems

Main cycle

```
unsigned short adc_value,display;
main()
{
  while(1) { display = adc_value }
}
```

Interrupt

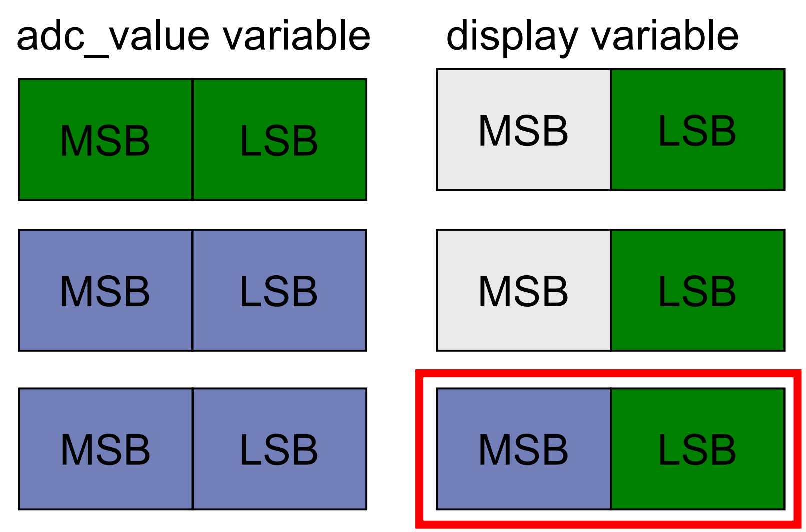
```
external unsigned short adc_value;
INTERRUPT(SIG_ADC )
{
 // Reading out the ADC values
  adc_value = read_adc();
}
```

Time

display = adc_value under execution (more assembly instructions)

ADC IT interrupts the main cycle

Remaining display = adc_value instructions executed



Problems II.: function reentrancy

- Typical shared resource problem
- Functions using global variables or hardware resources can not be used both form interrupt and main program
- Many compilers drop a warning for problematic situations

Embedded software architectures V.

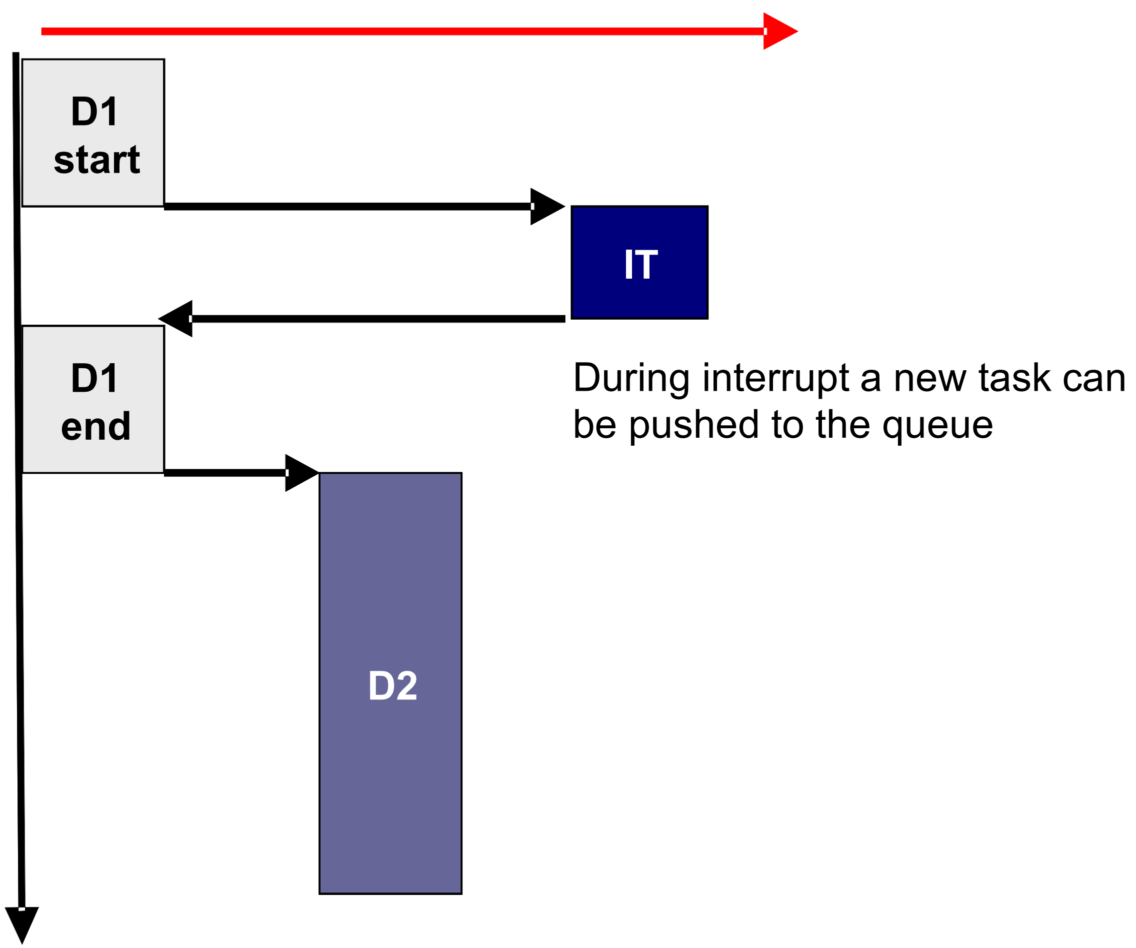
- Function queue based scheduling

```
void interrupt vDevice1(void)
{
    // Handle Device 1 time critical part
    // Put Device1_func to call queue
}
void interrupt vDevice2(void)
{
    // Handle Device 2 time critical part
    // Put Device2_func to call queue
}
void interrupt vDevice3(void)
{
    // Handle Device 3 time critical part
    // Put Device3_func to call queue
}
```

```
void main(void)
{
  while(1)
  {
     while(Function queue not empty)
            // Call first from queue
  }
}

void Device1_func (void)
{   // Handle Device 1   }

void Device2_func (void)
{   // Handle Device 2   }

void Device3_func (void)
{   // Handle Device 3  }
```

Embedded software architectures VI.

- Function queue based non-preemptive scheduling



Embedded software architectures VII.

- Function queue based non-preemptive scheduling
  - Can handle priorities
  - Shared resource problem present between the interrupt and the main cycle

  - Worst case response time for the highest priority job = response time of the longest job
  - Worst case response time for the highest priority job do not increase with the number of the jobs
  - Jitter can be low
  - New job do not modify significantly the timing of the higher priority jobs

Embedded software architectures VIII.

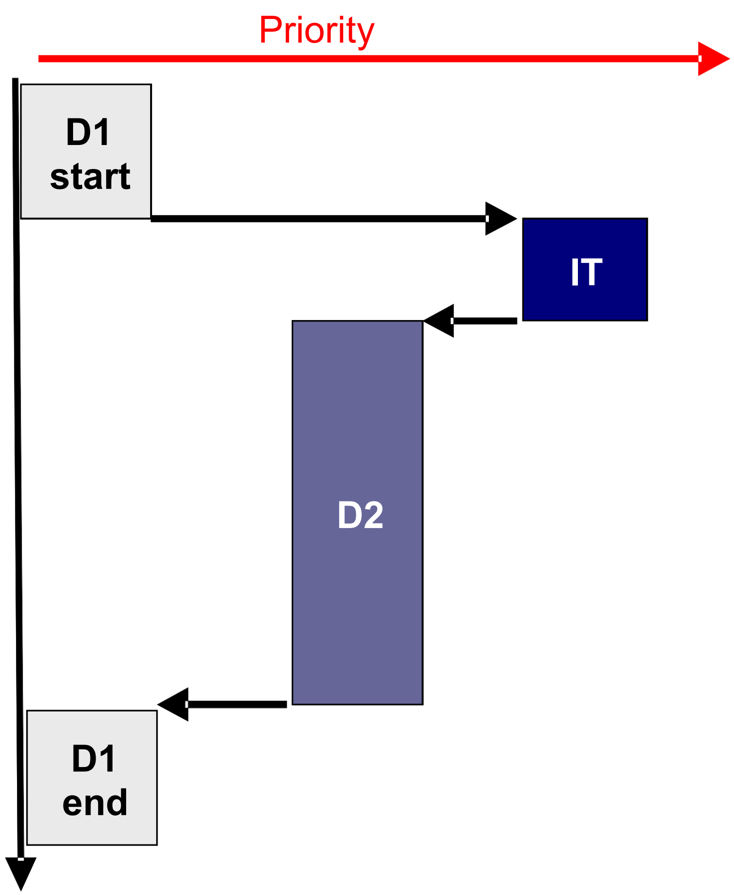
- Real Time OS, preemptive scheduling

```
void interrupt vDevice1(void)
{
   // Handle Device 1 time critical part
   // Set signal to Device1_task
}
void interrupt vDevice2(void)
{
   // Handle Device 2 time critical part
   // Set signal to Device2_task
}
void interrupt vDevice3(void)
{
   // Handle Device 3 time critical part
   // Set signal to Device3_task
}
```

```
void Device1_task (void)
{
  // Wait for signal to Device1_task
  // Handle Device 1
}

void Device2_task (void)
{
   // Wait for signal to Device2_task
   // Handle Device 2
}

void Device3_task (void)
{
   // Wait for signal to Device3_task
   // Handle Device 3
}
```

Embedded software architectures IX.

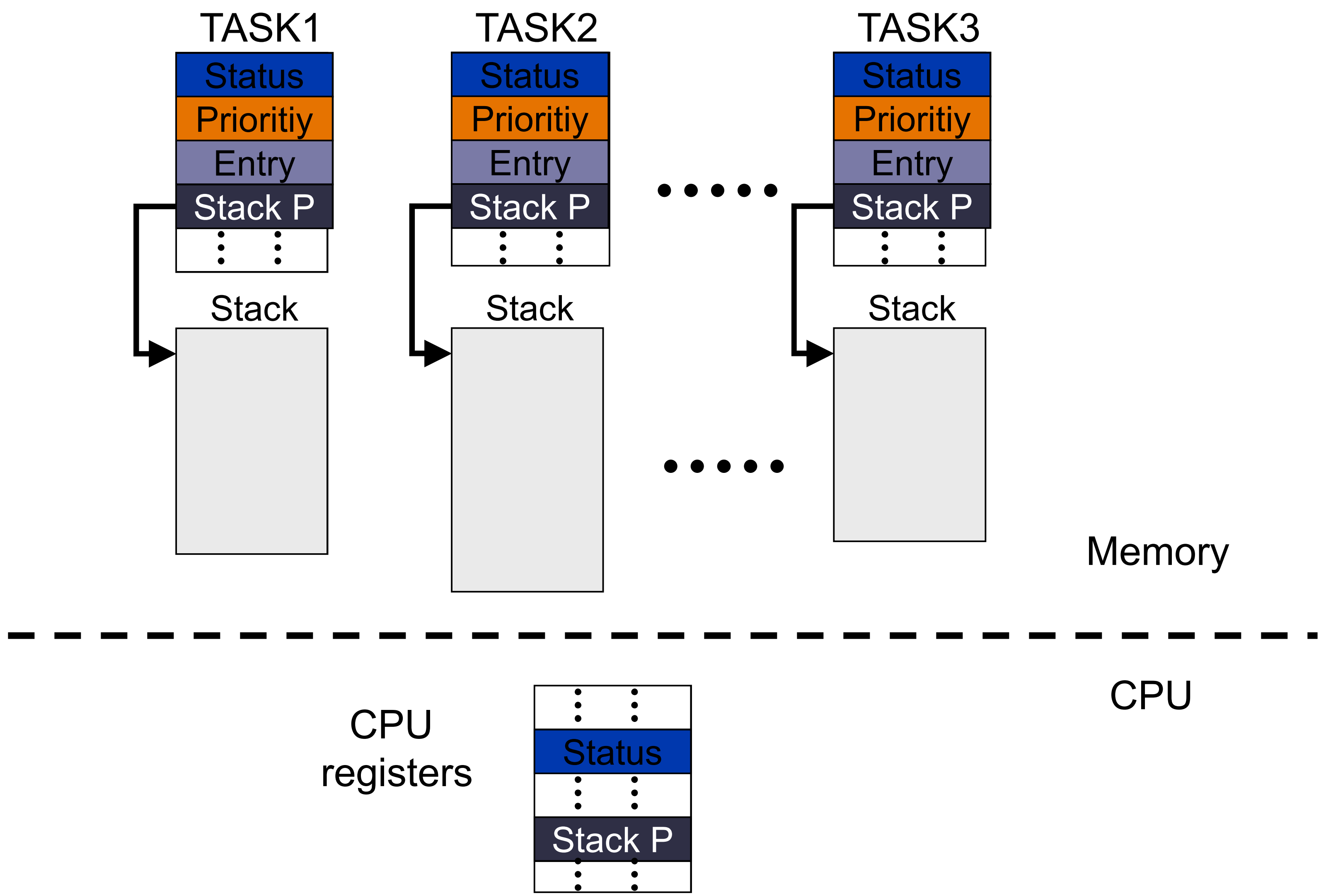
- Real Time OS, preemptive scheduling



Embedded software architectures X.

- Real Time OS, preemptive scheduling
  - Shared resource problem can present between the tasks and between tasks and interrupt
  - Worst case response time for the highest priority job = task switch time + IT
  - Worst case response time for the highest priority job do not increase with the number of the jobs
  - Jitter can be very low
  - New job do not modify the timing of the higher priority jobs

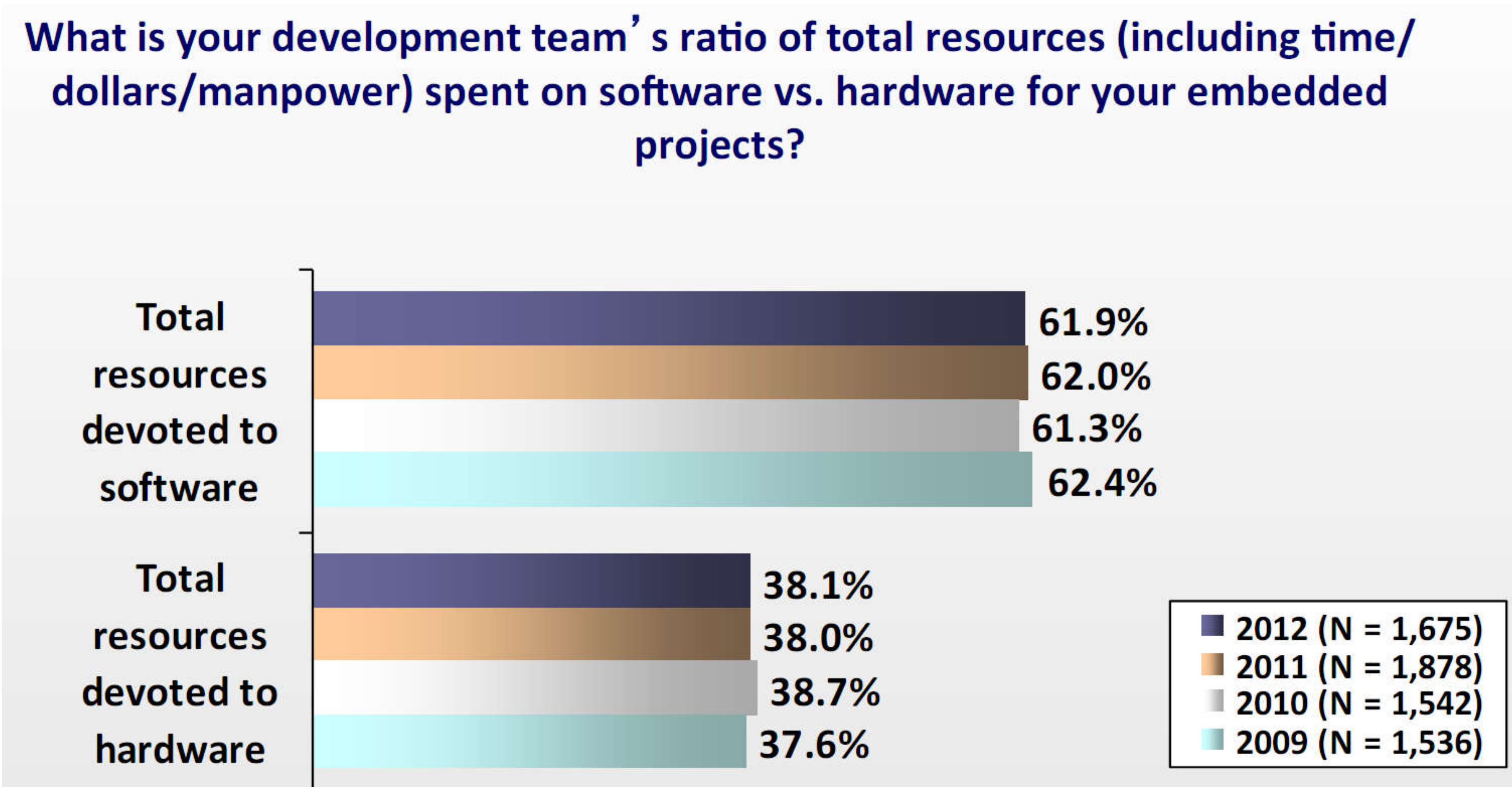
Task control, and task switching



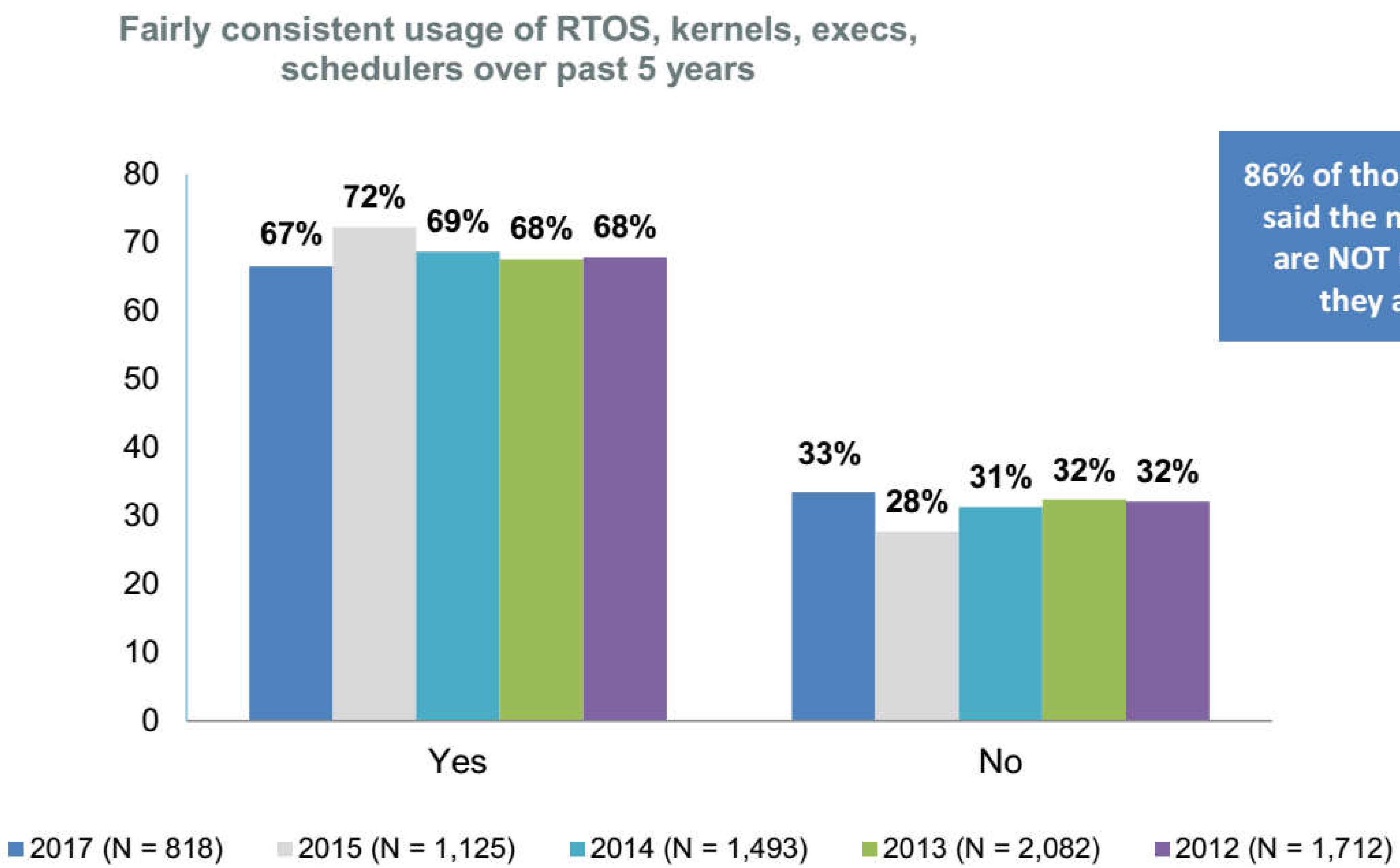
Comparing embedded OS and normal PC OS

- Footprint
- Configurability
- Real-time behavior
- The OS is started from the application. Not the OS starts the application
- There is no memory protection

Why operating systems are important?

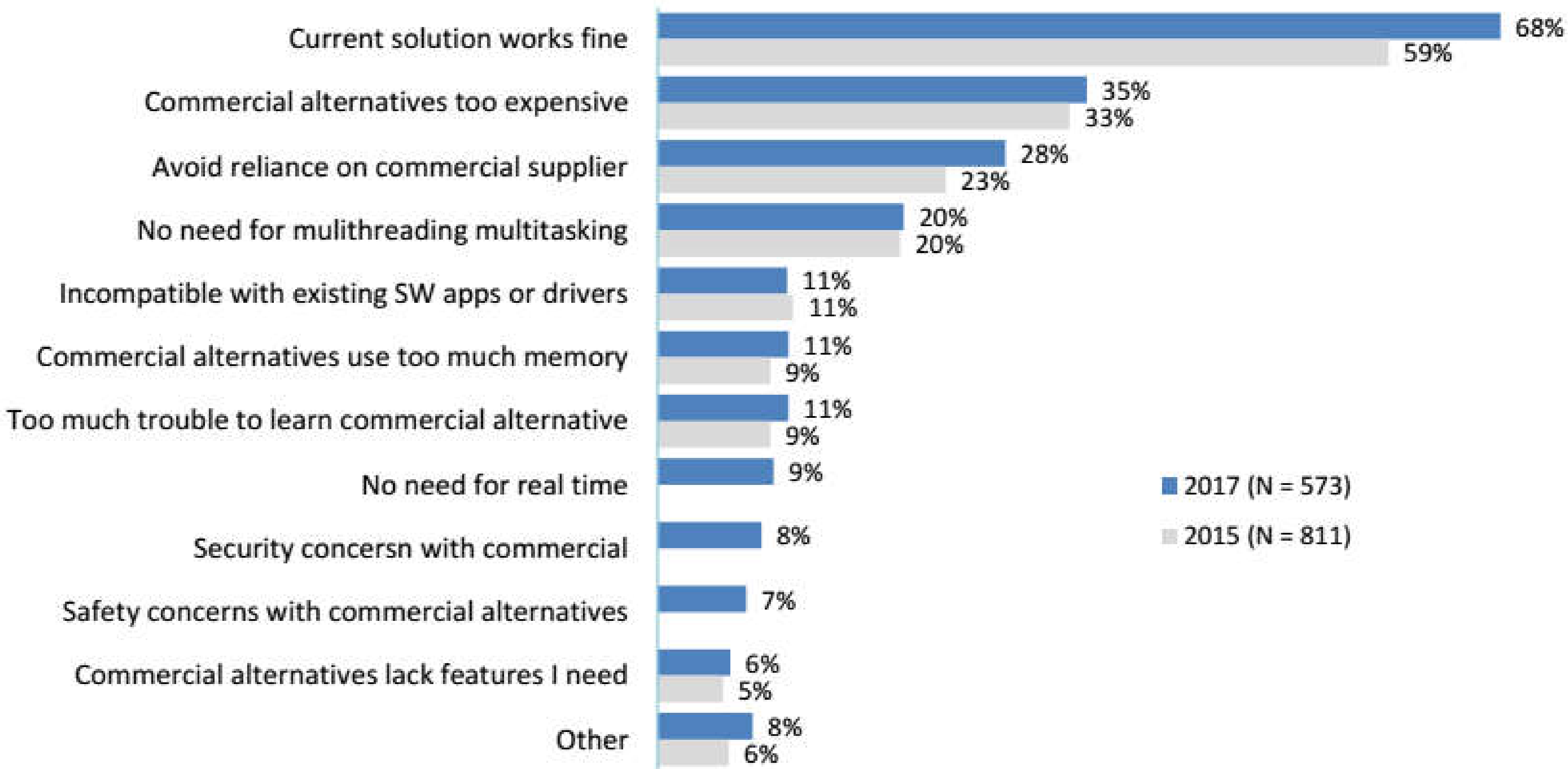


Usage statistics of embedded operating systems

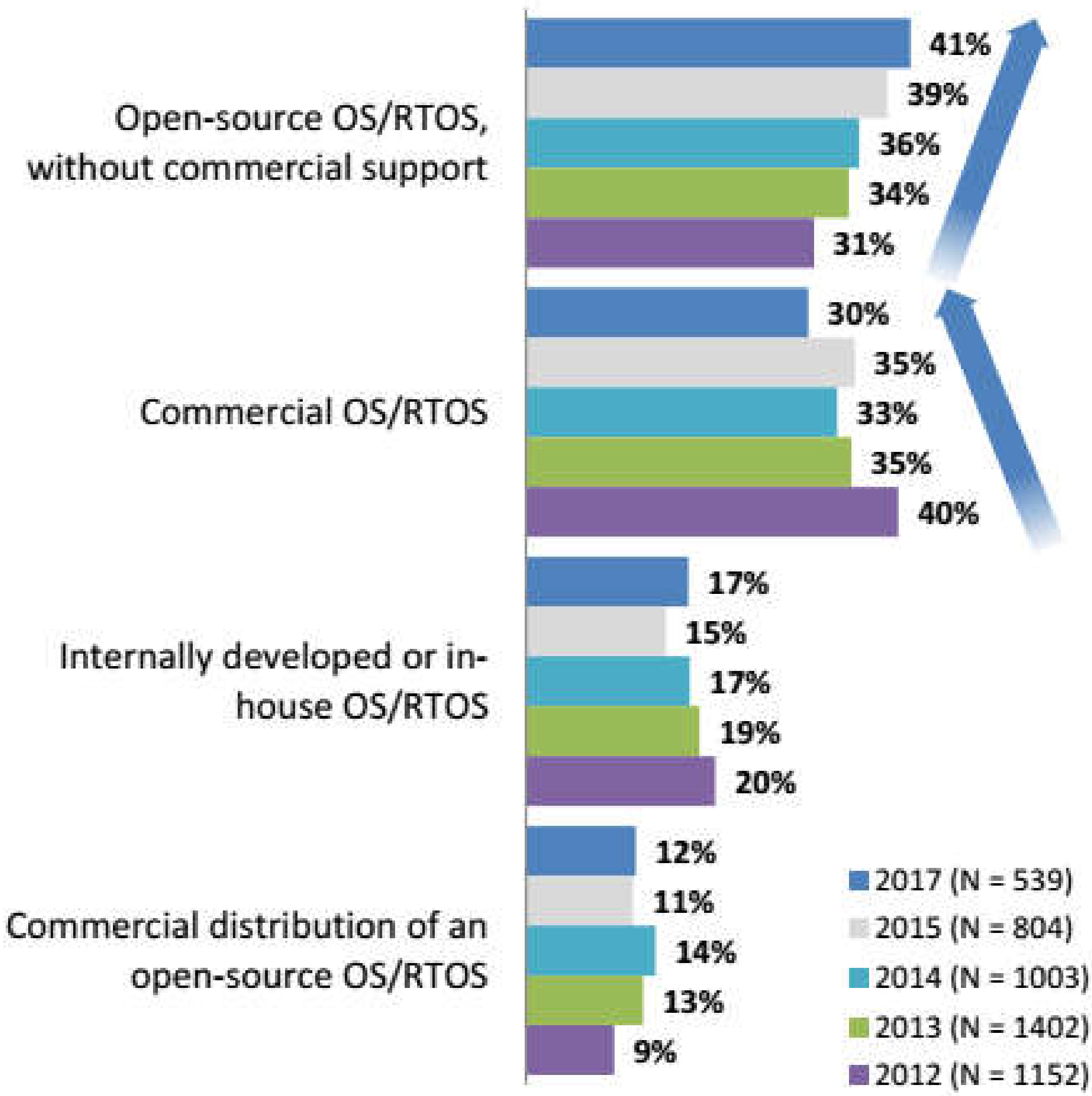


86% of those not using RTOSes, said the main reason RTOSes are NOT used is simply that they are not needed.

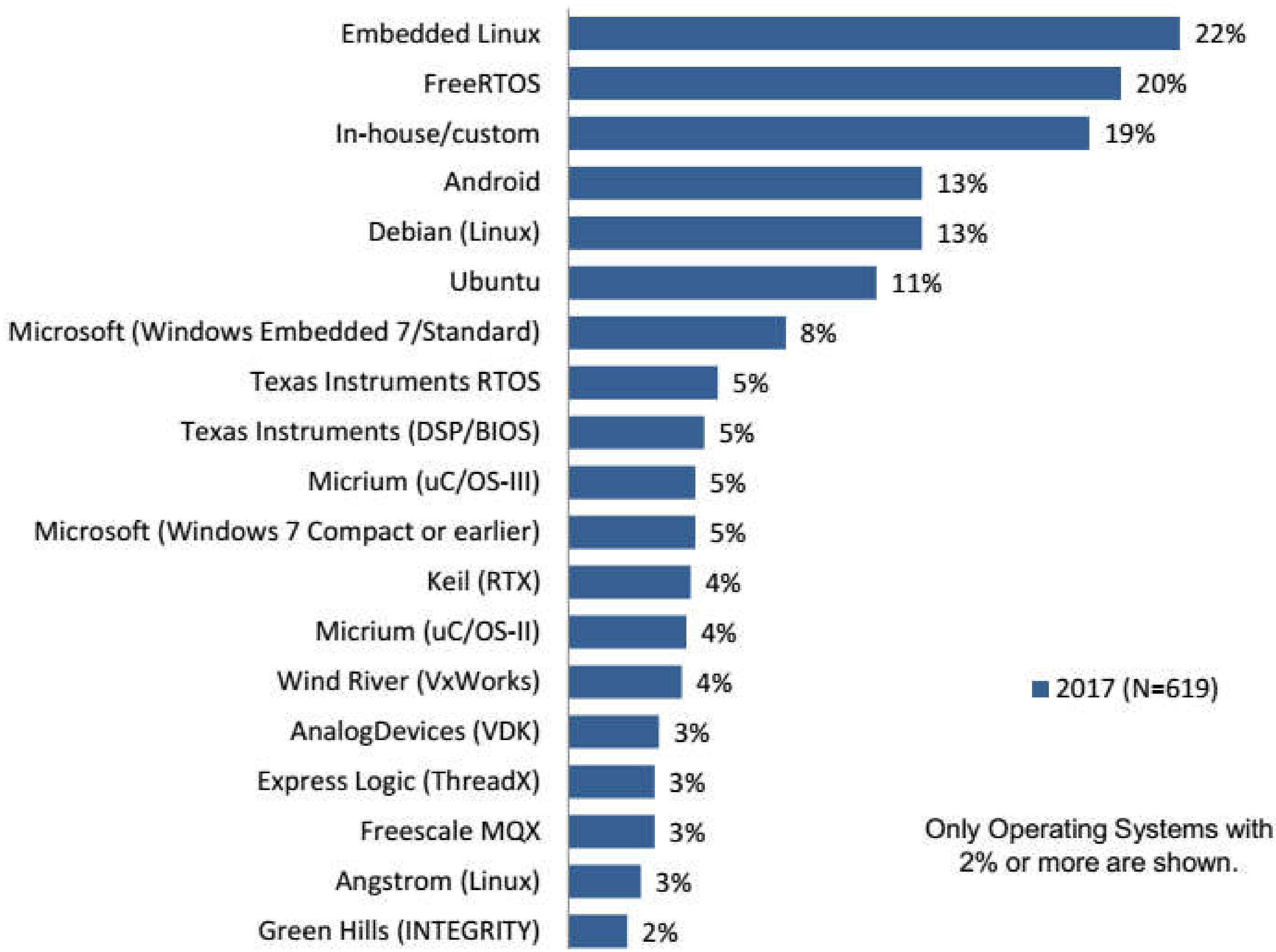
Why companies do not use OS



What are the trends in operating systems



Milyen OS-t használtak az elmúlt években?



μC-OS

μC/OS-II

The Real-Time Kernel

Micriμm

μC/OS-III™

The Real-Time Kernel

History of µC/OS

- **Jean J. Labrosse**

> ”Well, it can’t be that difficult to write a kernel. All it needs to do is save and restore processor registers.”

  - Most readed article of Embedded Systems Programming magazine in 199

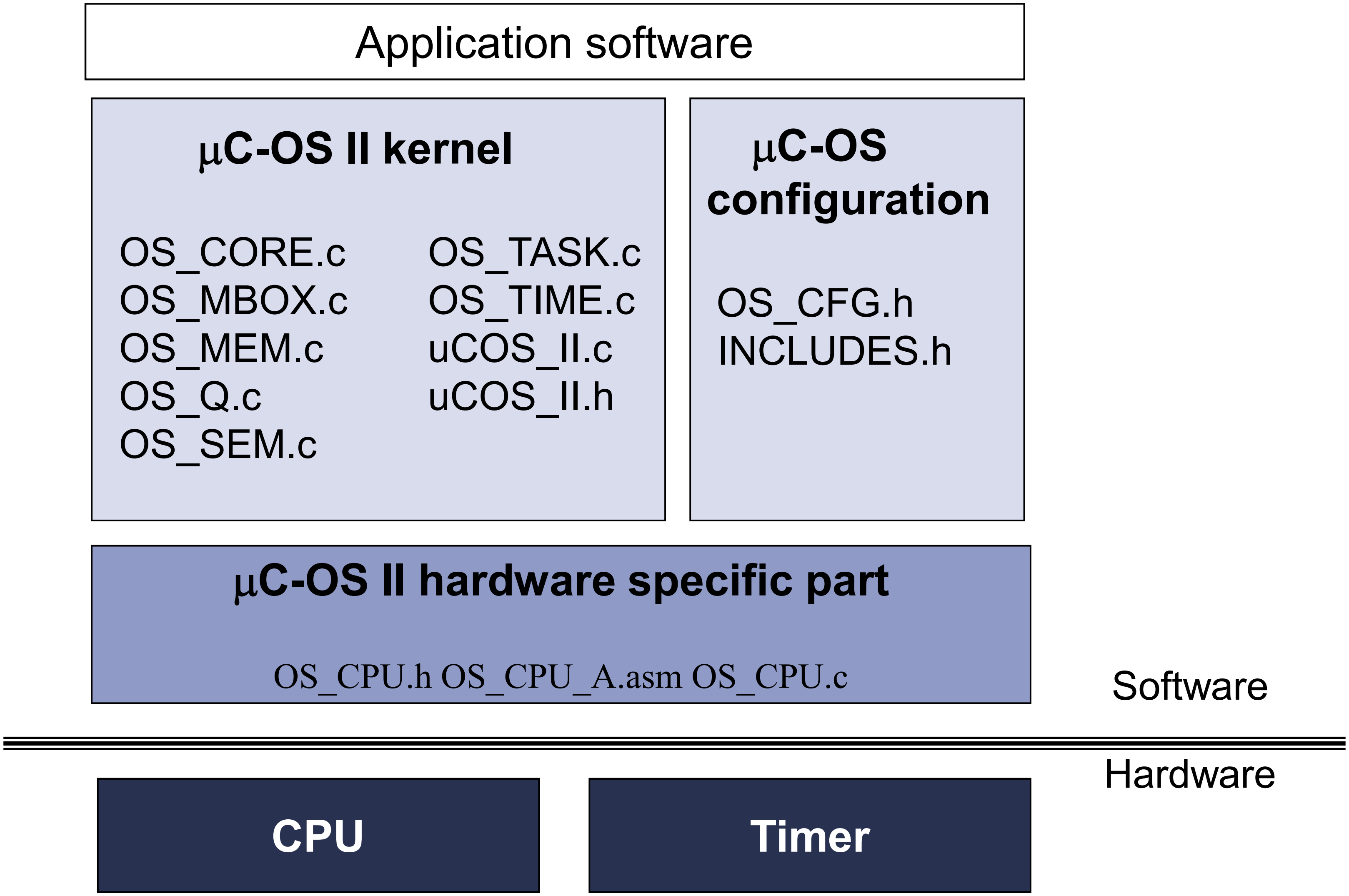
Properties of µC/OS

- Source code is available
- Easy to port
- Scalable
- multi-tasking
- preemptív scheduling
- Separate stack for every task
- Synchronization services: mailbox, queue, semaphore, timers etc.
- interrupt management

Properties of µC/OS

- Documentation available in a book (µC/OS-III, The Real-Time Kernel book with 300 pages)
- Kernel is free for educating purposes
- Supporting packages
  - TCP-IP (Protocol Stack)
  - FS (Embedded File System)
  - GUI (Embedded Graphical User Interface)
  - USB Device (Universal Serial Bus Device Stack)
  - USB Host (Universal Serial Bus Host Stack)
  - FL (Flash Loader)
  - Modbus (Embedded Modbus Stack)
  - CAN (CAN Protocol Stack)
  - BuildingBlocks (Embedded Software Components)
  - Probe (Real-Time Monitoring)

Architecture of µC/OS



Configuring µC/OS

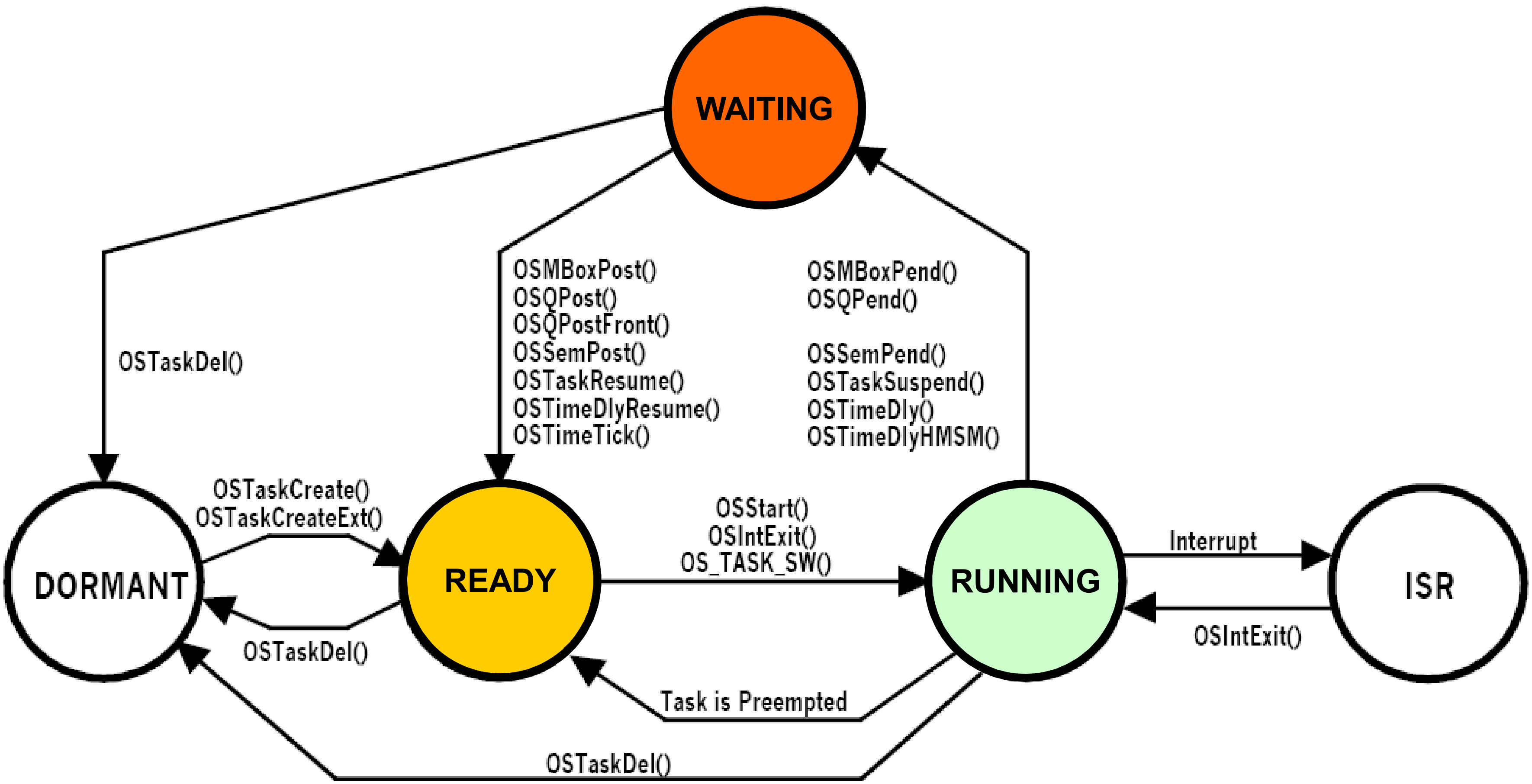
OS_CFG.h

```
                     /* -------------------- MESSAGE MAILBOXES --------------------- */
#define OS_MBOX_EN            1   /* Enable (1) or Disable (0) code generation for MAILBOXES  */
#define OS_MBOX_ACCEPT_EN     1   /*    Include code for OSMboxAccept()       */
#define OS_MBOX_DEL_EN        1   /*    Include code for OSMboxDel()          */
#define OS_MBOX_POST_EN       1   /*    Include code for OSMboxPost()         */
#define OS_MBOX_POST_OPT_EN   1   /*    Include code for OSMboxPostOpt()      */
#define OS_MBOX_QUERY_EN      1   /*    Include code for OSMboxQuery()        */
```

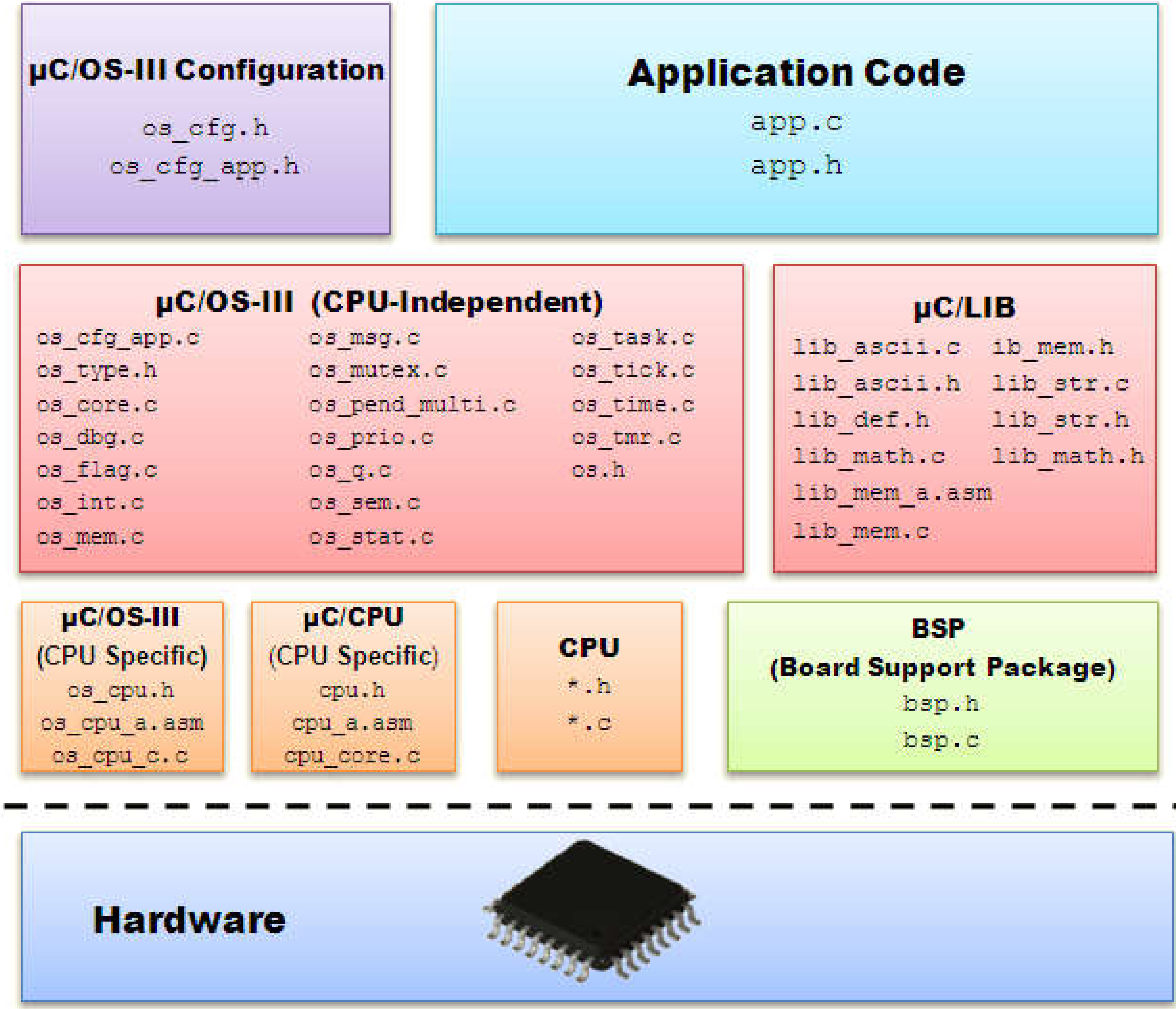
OS_MBOX.c

```
#if OS_MBOX_EN > 0
        .........
        #if OS_MBOX_ACCEPT_EN > 0
        .........
        #endif
        .........
        #if OS_MBOX_DEL_EN > 0
        .........
        #endif
#endif
```

Task states of µC/OS



Architecture of µC/OS



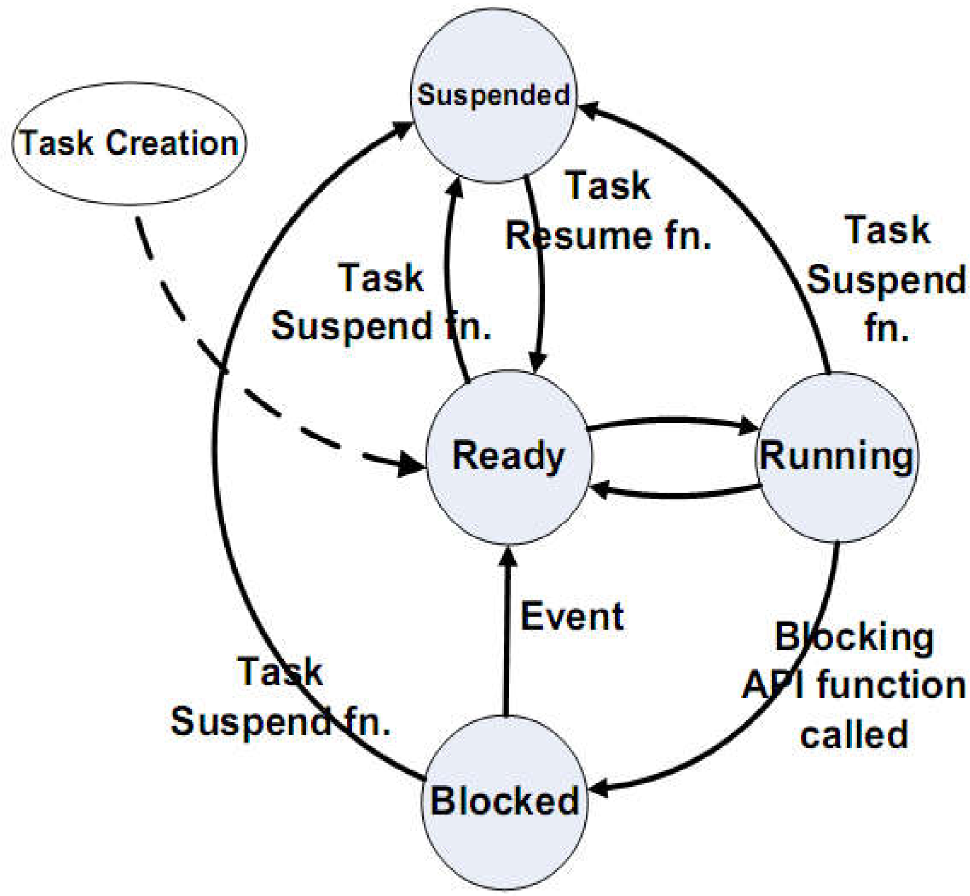
FreeRTOS

FreeRTOS

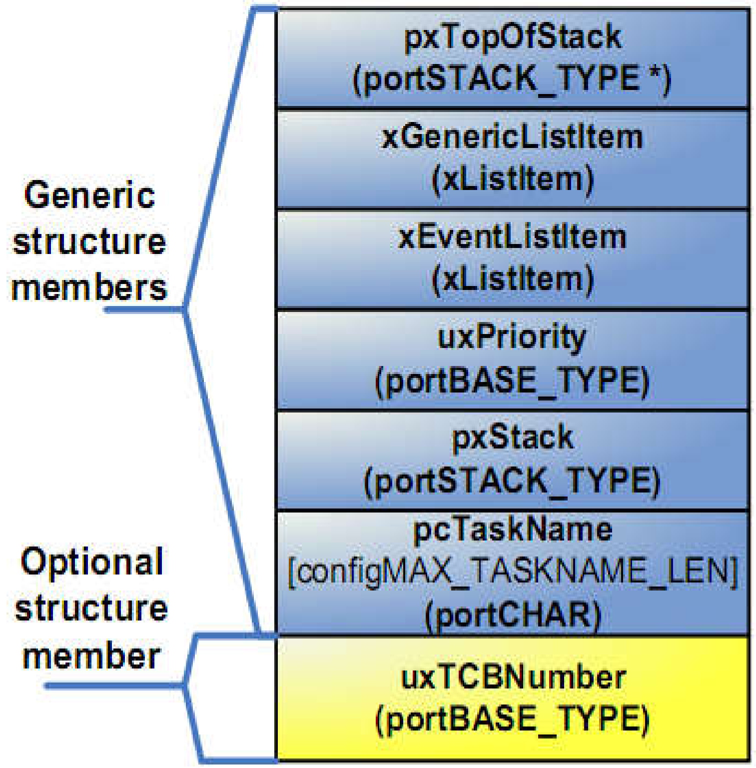
- Open source free kernel
  - www.freertos.org
- Most dynamic kernel of recent time
- Ports:
  - ARM7, ARM9, CortexM
  - Atmel AVR, AVR32
  - PIC18, PIC24, dsPIC, PIC32
  - Microblase…

FreeRTOS tasks

- Taszkok
  - Separate stack
  - High priority number means high priority
  - Idle task has priority 0



FreeRTOS task control block

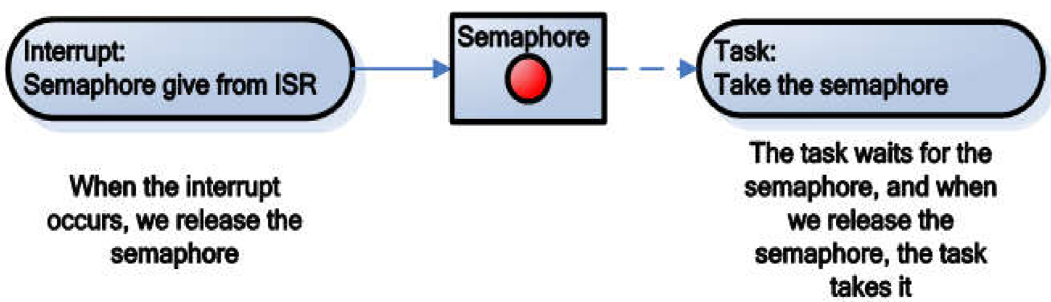


FreeRTOS task handling

```
void vOtherFunction( void )
{
xTaskHandle xHandle;

// Create the task, storing the handle.
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL,
    tskIDLE_PRIORITY, &xHandle );
// Use the handle to delete the task.
vTaskDelete( xHandle );
}
```

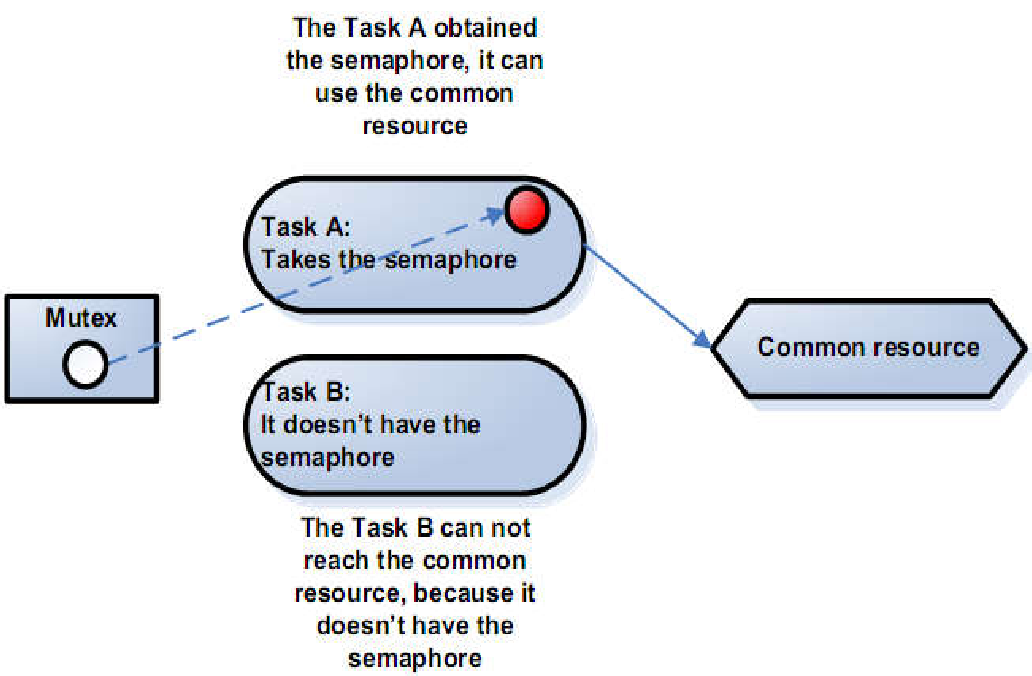
FreeRTOS task syncronisation

- Binary semaphores
  - vSemaphoreCreateBinary
  - xSemaphoreTake
  - xSemaphoreGive
  - xSemaphoreGiveFromISR
- Counting semaphores
  - Every semaphores has a number
  - Managing multiple identical resources
  - Event counting



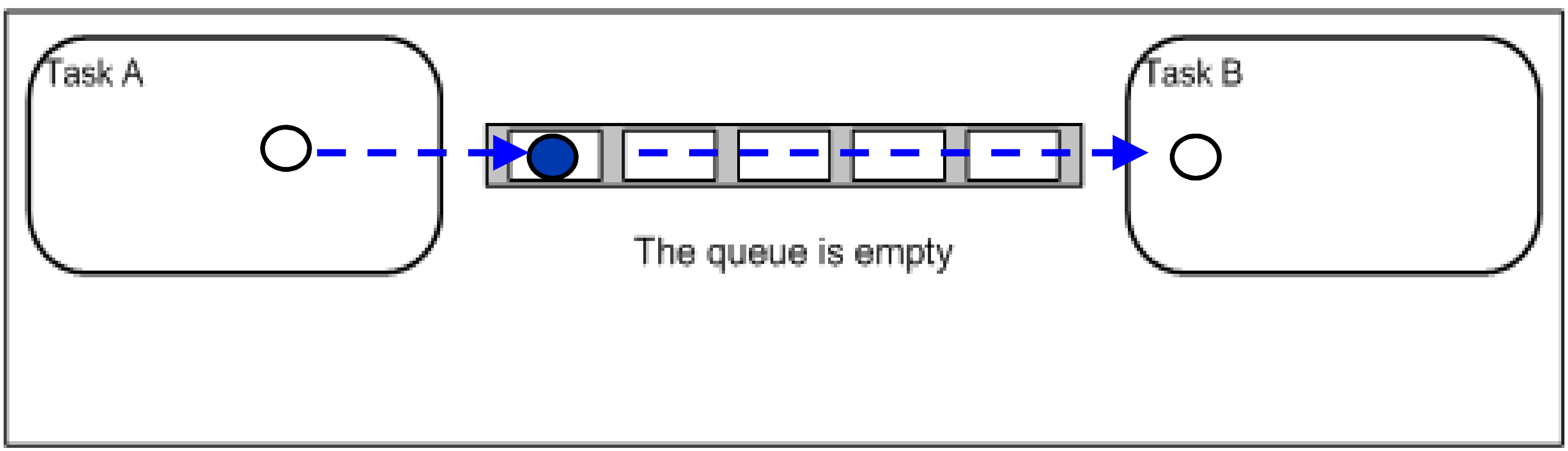
FreeRTOS task synchronisation

- Mutexes
  - Protected against priority inversion



FreeRTOS Queue

- Queue
  - Message sending between tasks
  - xQueueCreate
  - xQueueSend
  - xQueueReceive
  - xQueueSendFromISR



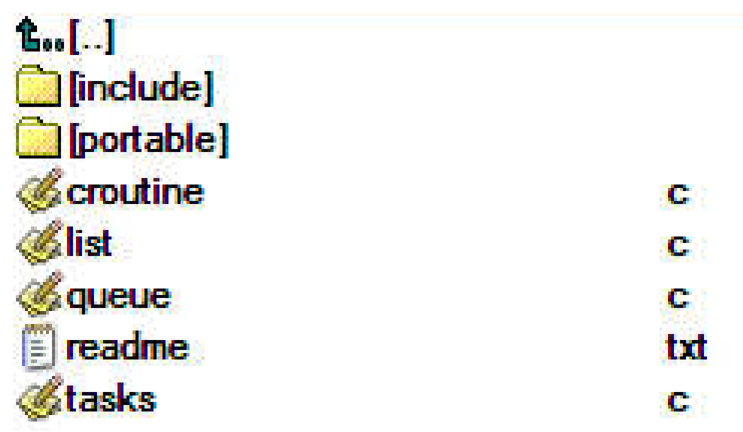
FreeRTOS CoRutin

- Simpler than tasks
- Non preemptive scheduling

- Sharing a stack between co-routines results in much lower RAM usage.
- Cooperative operation makes re-entrancy less of an issue.
- Very portable across architectures.
- Fully prioritised relative to other co-routines, but can always be preempted by tasks if the two are mixed.
- Lack of stack requires special consideration.
- Restrictions on where API calls can be made.
- Co-operative operation only amongst co-routines themselves.

FreeRTOS source code

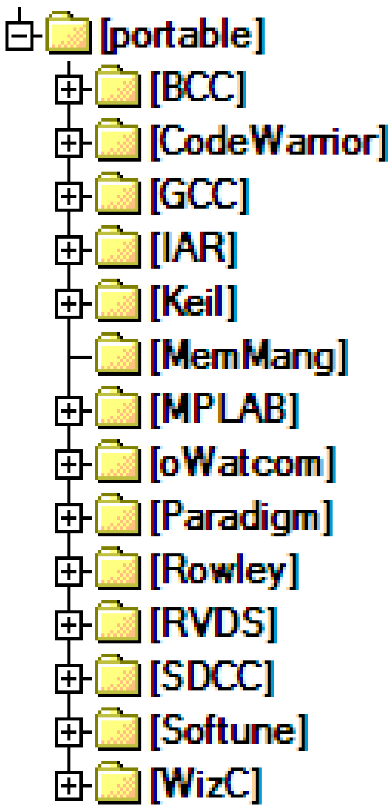
- Simple base kernel
  - tasks.c, queue.c, list.c



  - Containing task creation and synchronization mechanisms

Porting FreeRTOS

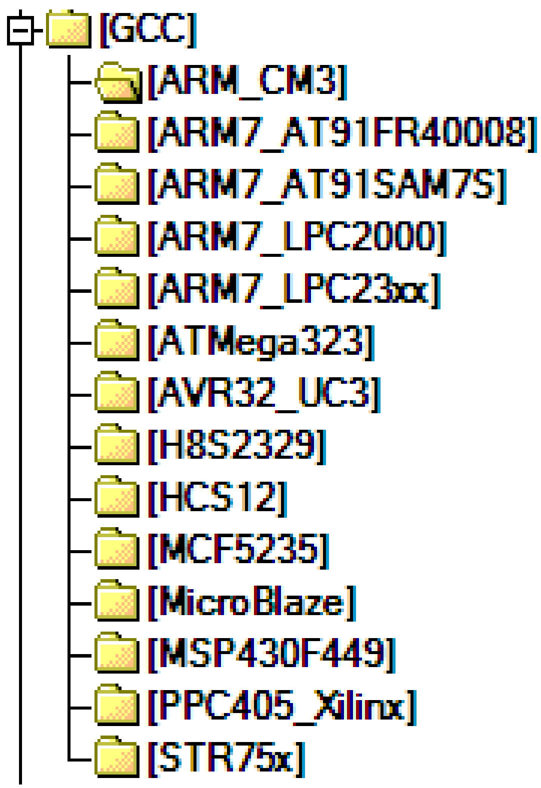
- Portable directory



- Task switching and Systick timer handling. Entering, exiting Critical section
- Organized based on toolchains

GCC specific parts

- GCC specific parts



- Sample port file

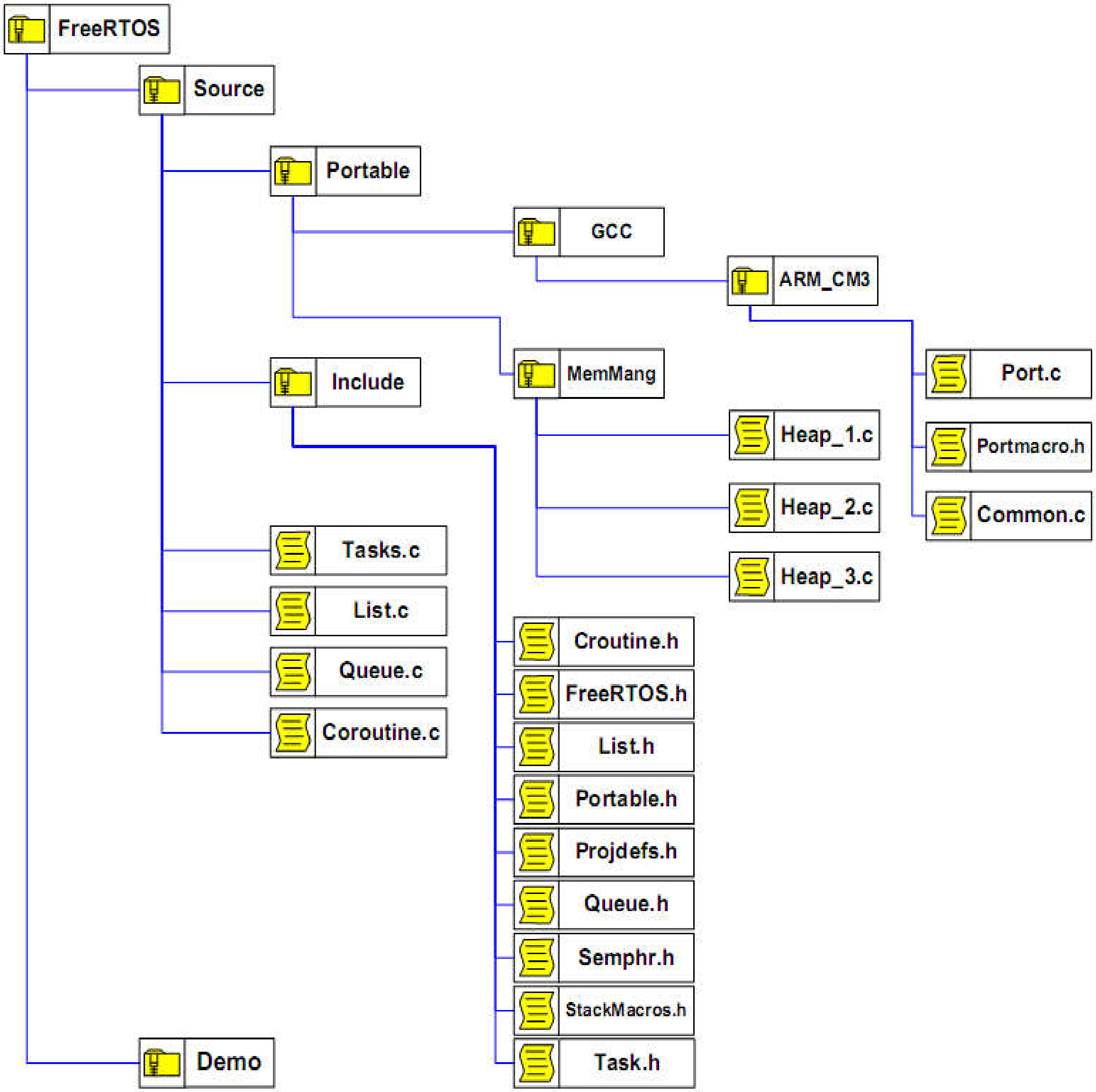
GCC demo projects

- Evaluation board and compiler specific parts

- [ARM7_LPC2106_GCC]
- [Common]
- [CORTEX_LM3S102_GCC]
- [CORTEX_LM3S102_KEIL]
- [CORTEX_LM3S102_Rowley]
- [CORTEX_LM3S316_IAR]
- [CORTEX_LM3S811_GCC]
- [CORTEX_LM3S811_IAR]
- [CORTEX_LM3S811_KEIL]
- [CORTEX_LM3Sxxxx_Eclipse]
- [CORTEX_LM3Sxxxx_IAR_Keil]
- [CORTEX_STM32F103_IAR]
- [CORTEX_STM32F103_Keil]
- [CORTEX_STM32F103_Primer_GCC]

- Startup code
- Evaluation board specific parts

FreeRTOS directory structure



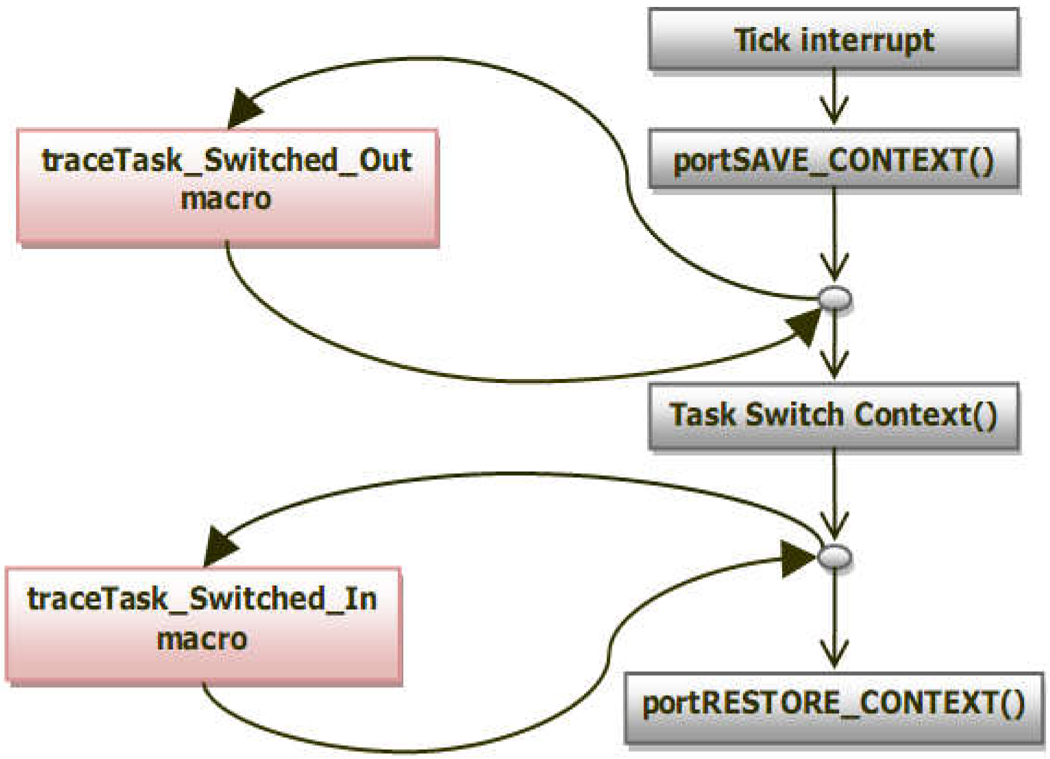
Configurating FreeRTOS

- FreeRTOS_Config.h

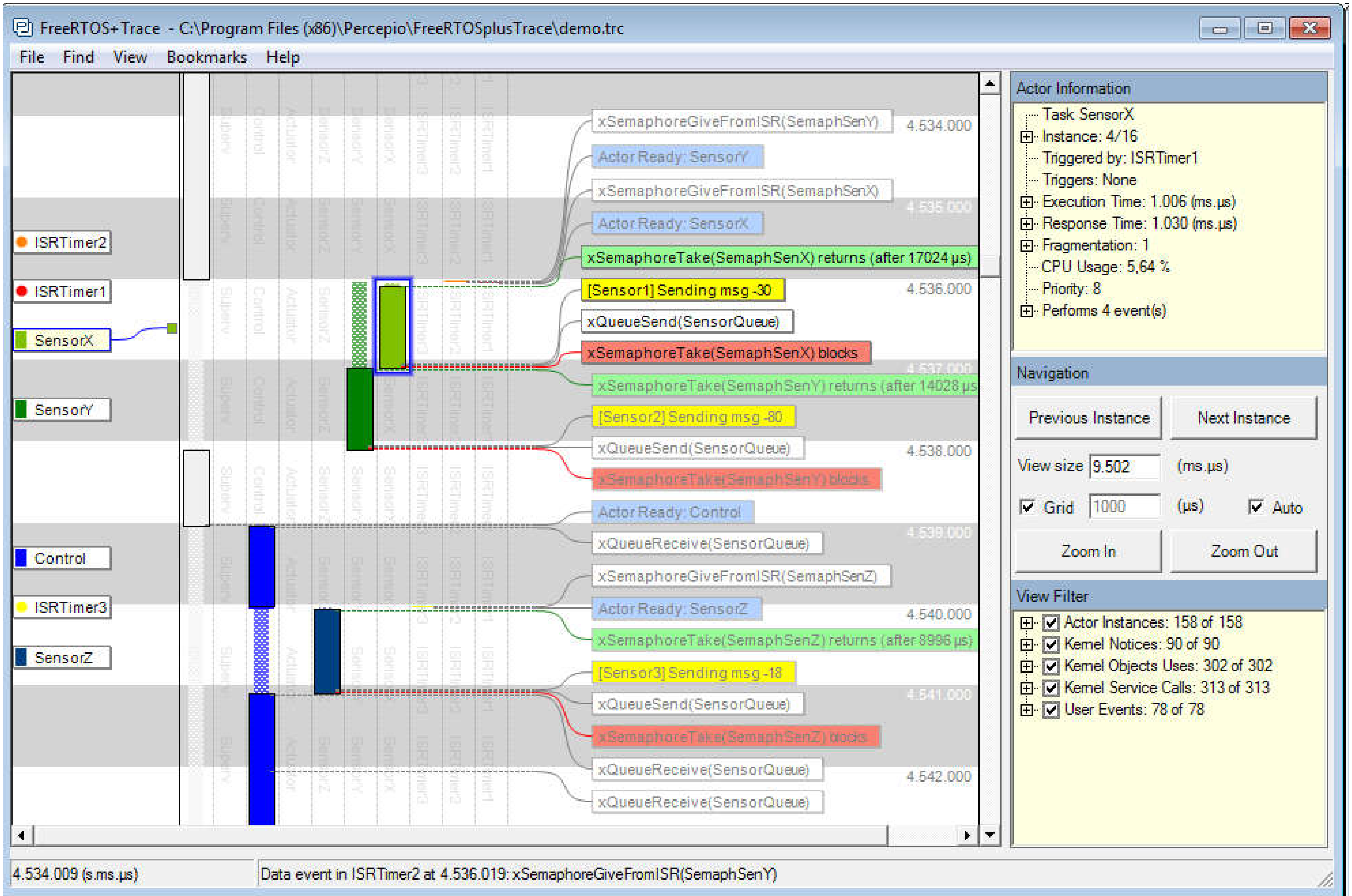
```
/*-----------------------------------------------------------
 * Application specific definitions.
 *
 * These definitions should be adjusted for your particular hardware and
 * application requirements.
 *
 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
 *----------------------------------------------------------*/

#define configUSE_PREEMPTION        1
#define configUSE_IDLE_HOOK         0
#define configUSE_TICK_HOOK         0
#define configCPU_CLOCK_HZ          ( ( unsigned portLONG ) 20000000 )
#define configTICK_RATE_HZ          ( ( portTickType ) 1000 )
#define configMINIMAL_STACK_SIZE    ( ( unsigned portSHORT ) 70 )
#define configTOTAL_HEAP_SIZE       ( ( size_t ) ( 7000 ) )
#define configMAX_TASK_NAME_LEN     ( 10 )
#define configUSE_TRACE_FACILITY    0
#define configUSE_16_BIT_TICKS      0
#define configIDLE_SHOULD_YIELD     0
#define configUSE_CO_ROUTINES       0

#define configMAX_PRIORITIES        ( ( unsigned portBASE_TYPE ) 5 )
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )
```

Trace hooks

- Every part of the kernel can be instrumented



Trace application

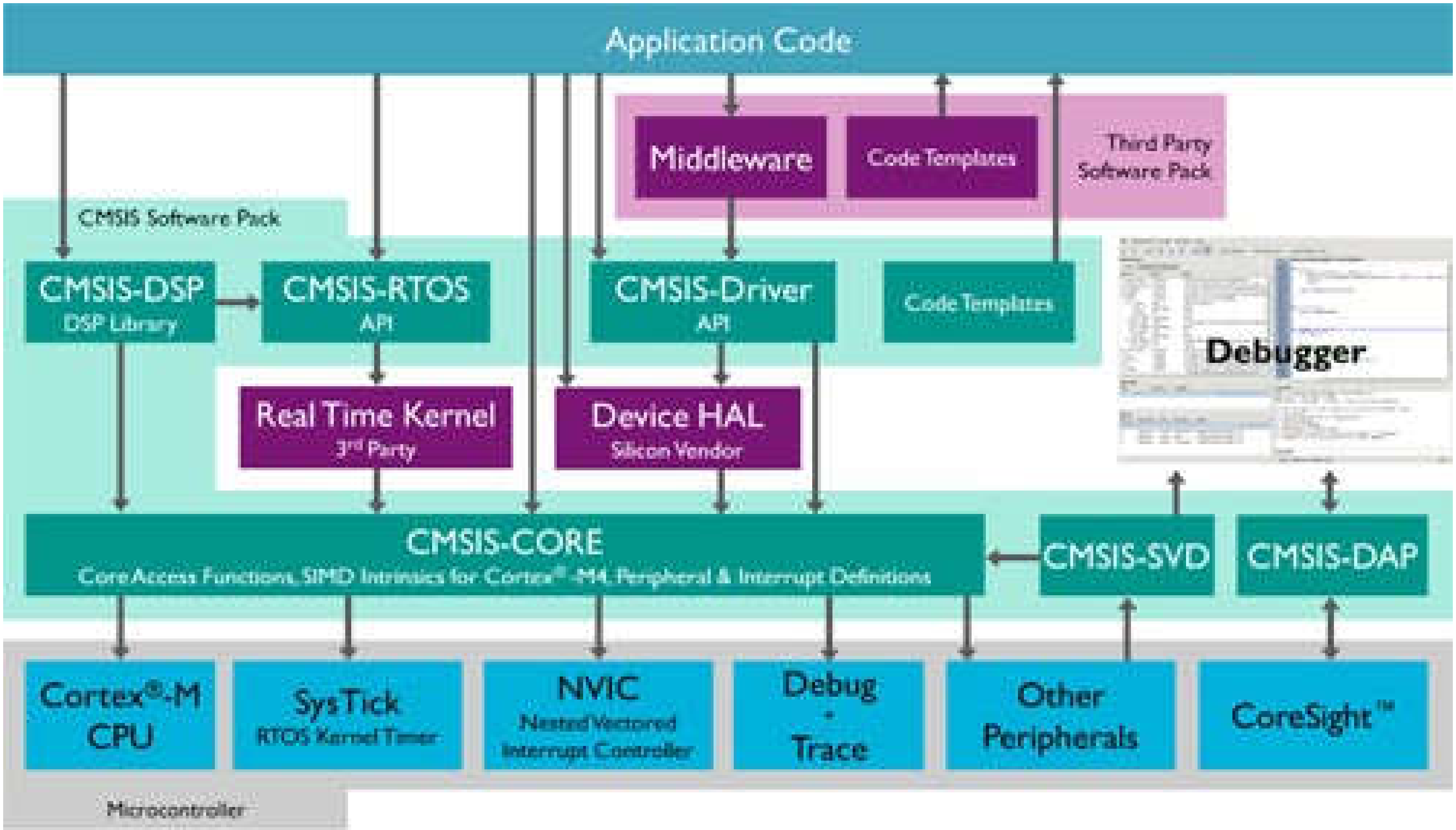


FreeRTOS commercial version

- OpenRTOS
  - Commercial version
  - USB, File system, TCP-IP support
- SafeRTOS
  - SIL3 certificate
  - Integrated into Stellaris LM3S9B96 ROM

CMSIS RTOS

- RTOS abstraction layer



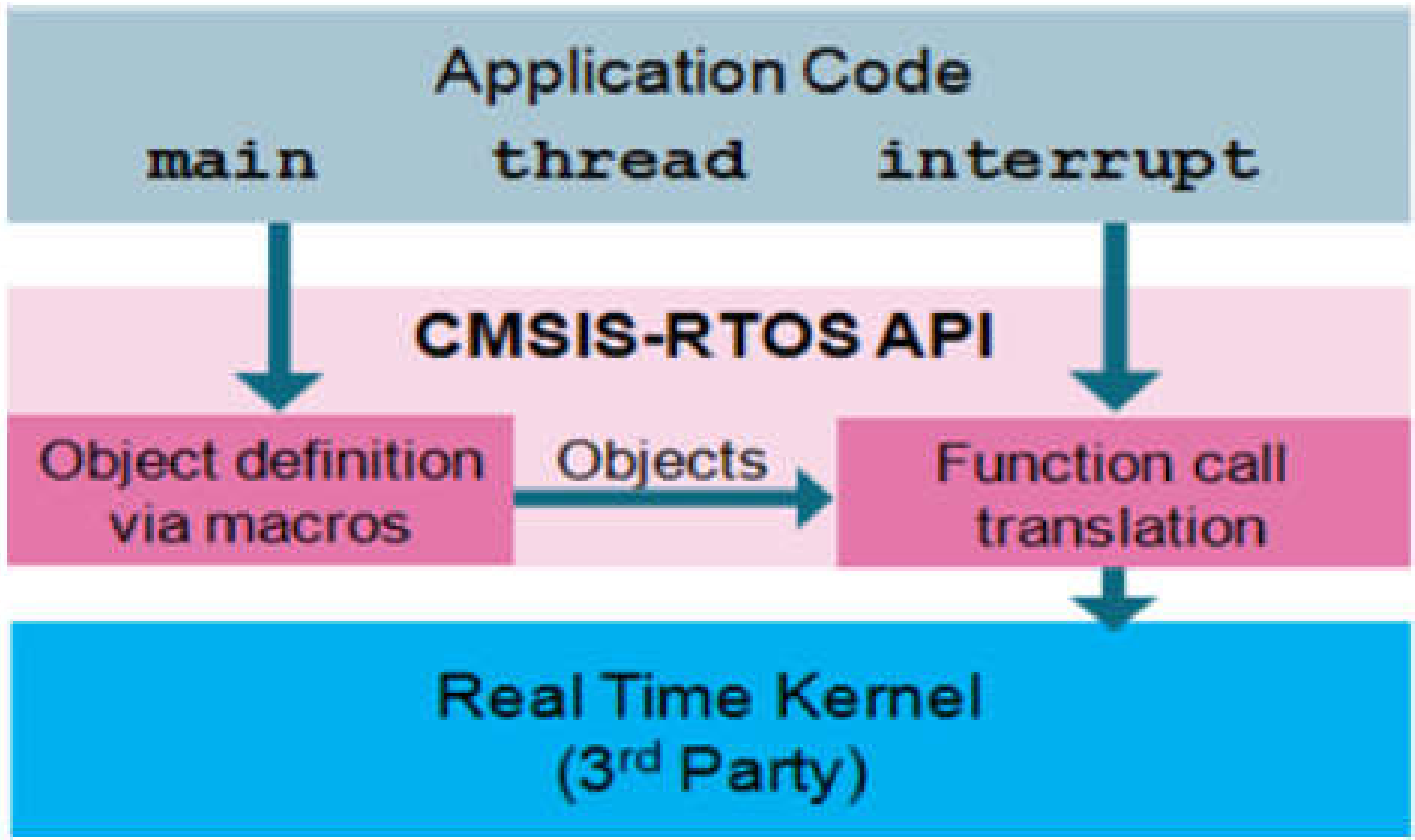
CMSIS RTOS

- RTOS abstraction
  - Thread handling function
  - Synchronization functions

- Supported by more and more platforms
  - Keil MDK
  - STM32 Cube
  - Mbed

CMSIS RTOS

- Architecture



CMSIS RTOS

- Kernel handling

osStatus	**osKernelInitialize** (void) Initialize the RTOS Kernel for creating objects.
osStatus	**osKernelStart** (void) Start the RTOS Kernel.
int32_t	**osKernelRunning** (void) Check if the RTOS kernel is already started.
uint32_t	**osKernelSysTick** (void) Get the RTOS kernel system timer counter.

CMSIS RTOS

- Thread management

osThreadId	**osThreadCreate** (const **osThreadDef_t** *thread_def, void *argument) Create a thread and add it to Active Threads and set it to state READY.
osThreadId	**osThreadGetId** (void) Return the thread ID of the current running thread.
osStatus	**osThreadTerminate** (**osThreadId** thread_id) Terminate execution of a thread and remove it from Active Threads.
osStatus	**osThreadSetPriority** (**osThreadId** thread_id, **osPriority** priority) Change priority of an active thread.
osPriority	**osThreadGetPriority** (**osThreadId** thread_id) Get current priority of an active thread.
osStatus	**osThreadYield** (void) Pass control to next thread that is in state **READY**.

CMSIS RTOS

- Delaying functuins

osStatus	**osDelay** (uint32_t millisec) Wait for Timeout (Time Delay).
osEvent	**osWait** (uint32_t millisec) Wait for Signal, Message, Mail, or Timeout.

- Timing functions

osTimerId	**osTimerCreate** (const osTimerDef_t *timer_def, os_timer_type type, void *argument) Create a timer.
osStatus	**osTimerStart** (osTimerId timer_id, uint32_t millisec) Start or restart a timer.
osStatus	**osTimerStop** (osTimerId timer_id) Stop the timer.
osStatus	**osTimerDelete** (osTimerId timer_id) Delete a timer that was created by osTimerCreate.

CMSIS RTOS

- Synchronization functions
  - Signal events
  - Semaphores
  - Mutex
  - Message queue
  - Mail queue
  - Memory Pool