Politechnika Łódzka

Instytut Elektroniki

Systemy Zintegrowane (ang. *Embedded Systems*)

Real Time System Concepts

mgr inż. Paweł Poryzała

Zakład Elektroniki Medycznej

Źródła

- Do przygotowania prezentacji wykorzystano materiały z poniższych:
  - FreeRTOS.org^{TM} Project – Designed for Microcontrollers
    - http://www.freertos.org/
  - uC/OS – The Real Time Kernel, Jean. J. Labrosse
  - Real-Time Concepts for Embedded Systems, Qing Li and Carolyn Yao
  - http://www.sics.se/~adam/pt/ - Protothreads
  - www.embedded.com

Embedded System

- computing systems with tightly coupled hardware and software integration, that are designed to perform a dedicated function, for eg.
  - router
  - printer
  - set top box
  - mp3 player
  - navigation system
  - etc.
- „embedded” – systems are usually an integral part of a larger system, known as the embedding system. Multiple embedded systems can coexist in an embedding system

Event Driven Embedded Systems

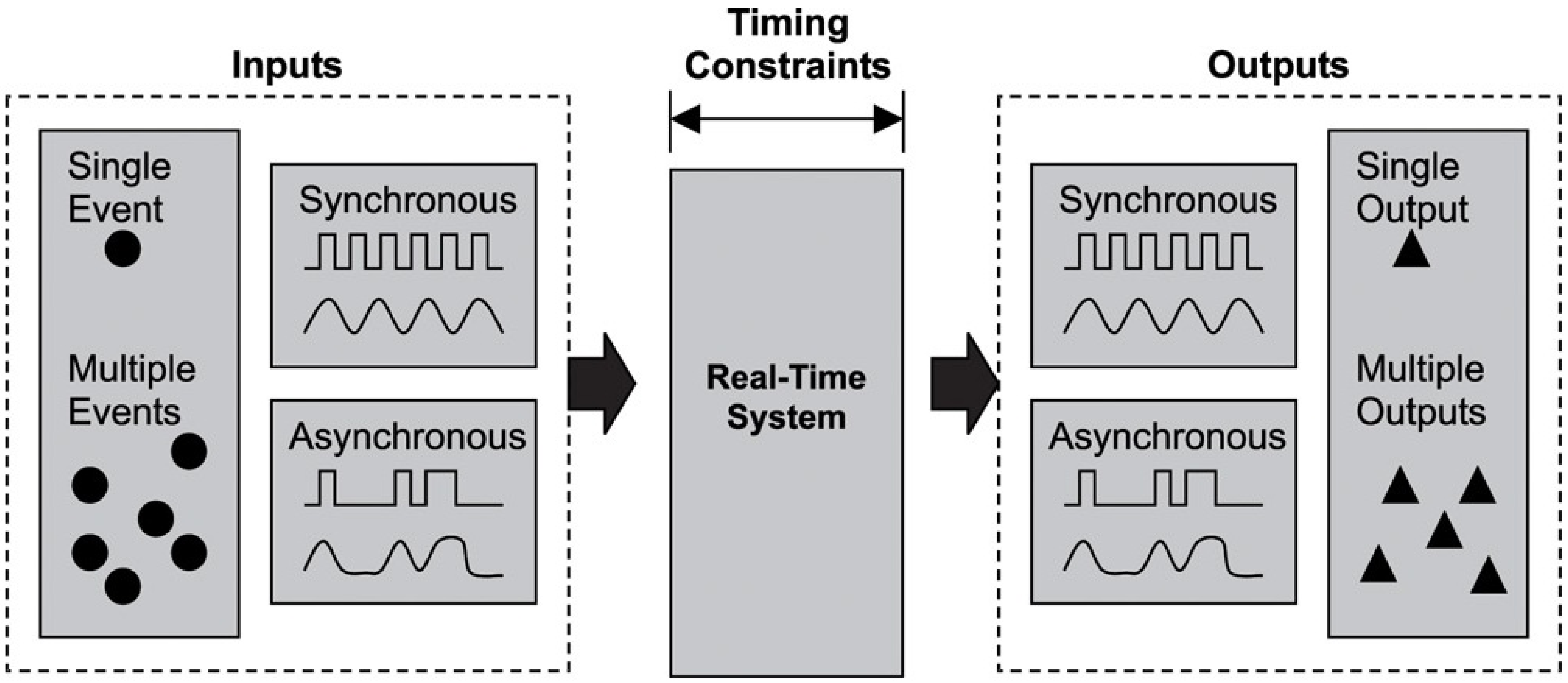
- event-driven embedded system:
  - wait for some event (a time tick, a button press, a mouse click, or the arrival of a data packet)
  - recognize the event
  - react by performing the appropriate computation. This reaction might include:
    - manipulating the hardware,
    - generating secondary, "soft" events that trigger other internal software components.
  - if event-handling action is complete enter a dormant state in anticipation of the next event.

- event-driven embedded system + time constraints =

  real-time embedded system

Real-Time Embedded System

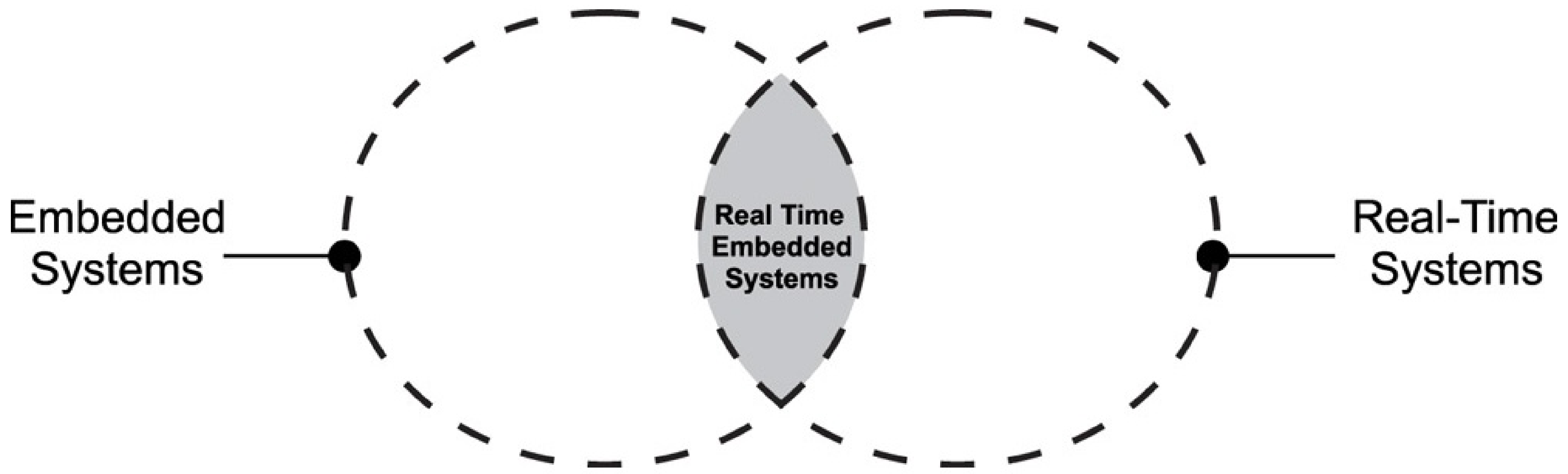
- system that responds to external events in a timely fashion



- responding to external events includes:
  - recognizing when an event occurs,
  - performing the required processing as a result of the event,
  - outputting the necessary results within a given time constraint.

Real-Time Embedded System

- not all embedded systems exhibit real-time behaviors nor are all real-time systems embedded



Strategie programowania

- *Super Loop*,
  - nazywane również *Infinite/Endless Loop, Foreground/Background Systems* etc.
- State Machines (maszyny stanu)
- Protothreads (*protowątki*) lub rozwiązania typu Super Simple Tasker (SST)
- RTOS
  - kooperatywny
  - wywłaszczeniowy

Super Loop

- *Super Loop* jest konieczna ponieważ… nie mamy systemu operacyjnego do którego można wrócić…

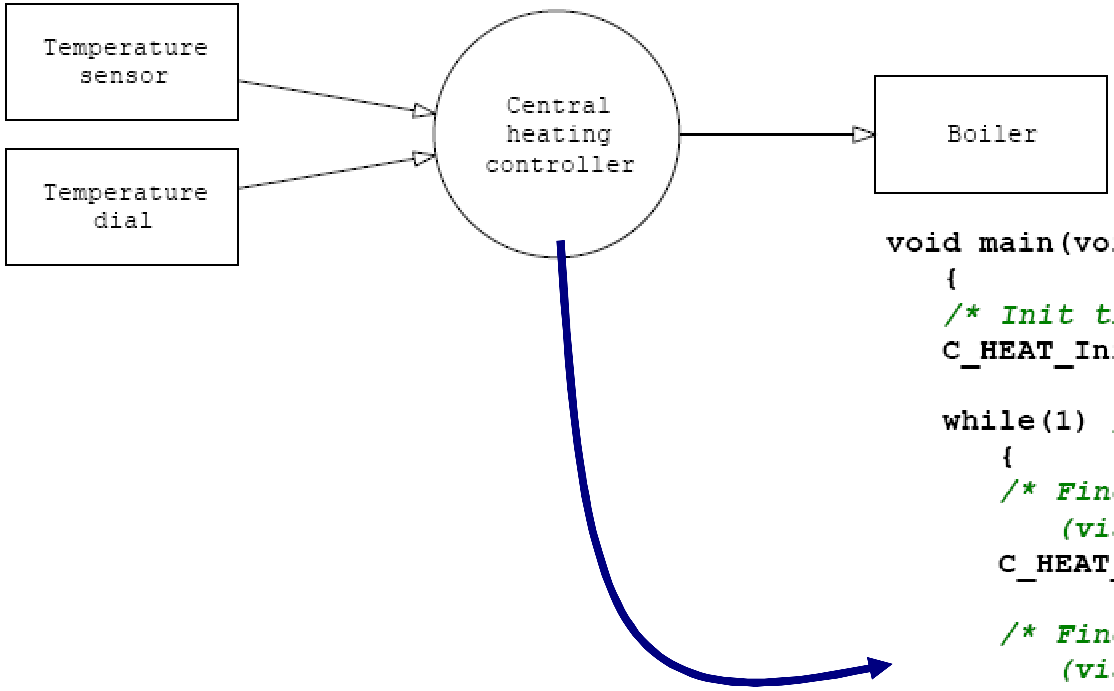
```c
void main(void)
   {
   /* Prepare for Task X */
   X_Init();

   while(1) /* 'for ever' (Super Loop) */
      {
      X();  /* Perform the task */
      }
   }
```

Super Loop

- Zalety:
  - Prostota – budowa, testowanie oraz późniejsza rozbudowa,
  - Wysoka wydajność oraz brak wymagań dotyczących sprzętu na którym uruchomiona ma być aplikacja,
  - Przenoszenie na inne platformy.
- Wady:
  - Bardzo proste aplikacje,
  - Aplikacje wymagające spełnienia określonych, krytycznych ram czasowych – np. proces próbkowania z częstotliwością 500Hz,
  - Nieustanna praca z „pełną mocą” (tryb normalny) – problem, gdy aplikacja wymaga oszczędnego zarządzania energią.

Super Loop

- Przykład:



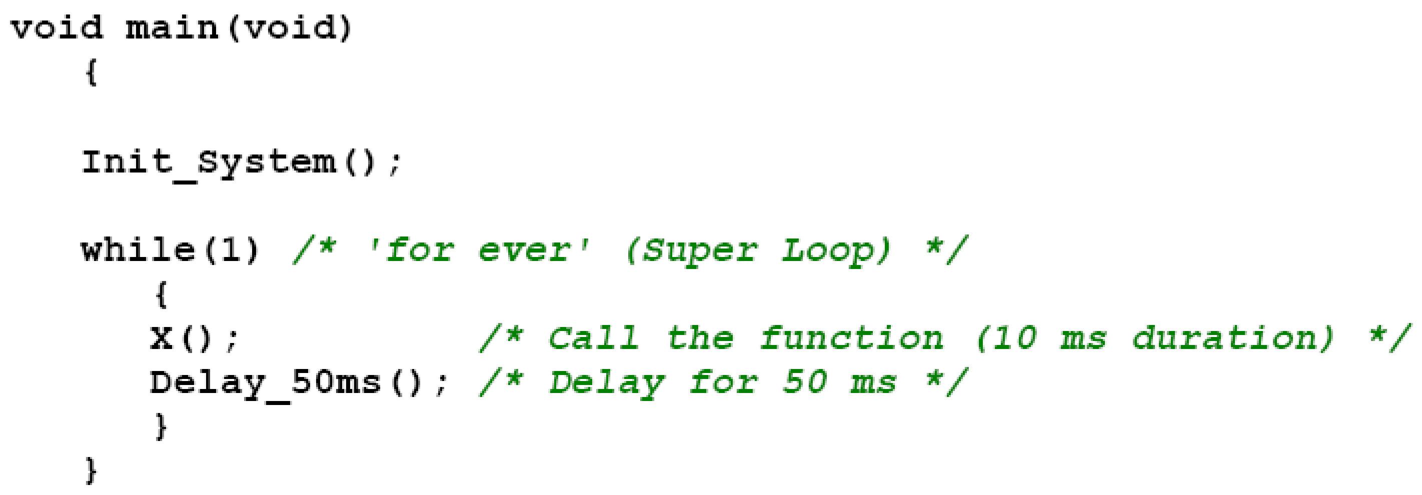
```
void main(void)
   {
   /* Init the system */
   C_HEAT_Init();

   while(1) /* 'for ever' (Super Loop) */
      {
      /* Find out what temperature the user requires
         (via the user interface) */
      C_HEAT_Get_Required_Temperature();

      /* Find out what the current room temperature is
         (via temperature sensor) */
      C_HEAT_Get_Actual_Temperature();

      /* Adjust the gas burner, as required */
      C_HEAT_Control_Boiler();
      }
   }
```

Michael J. Pont.

Super Loop: wymagania współczesnych aplikacji

- Wymagania stawiane współczesnym urządzeniom (systemom zintegrowanym):
  - Prędkość pojazdu ma być mierzona co 0,5s,
  - Wyświetlacz powinien być odświeżany 40 razy na sekundę,
  - Obliczone położenie przepustnicy musi zostać ustawione co 0,5s,
  - Wibracje silnika muszą być mierzone (próbkowane) z częstotliwością 1000Hz,
  - Reakcja na przerwanie czujnika krańcowego nie może przekroczyć 200ms,
  - Klawiatura musi być skanowana co 50ms.

Super Loop: sposoby rozwiązania problemu…

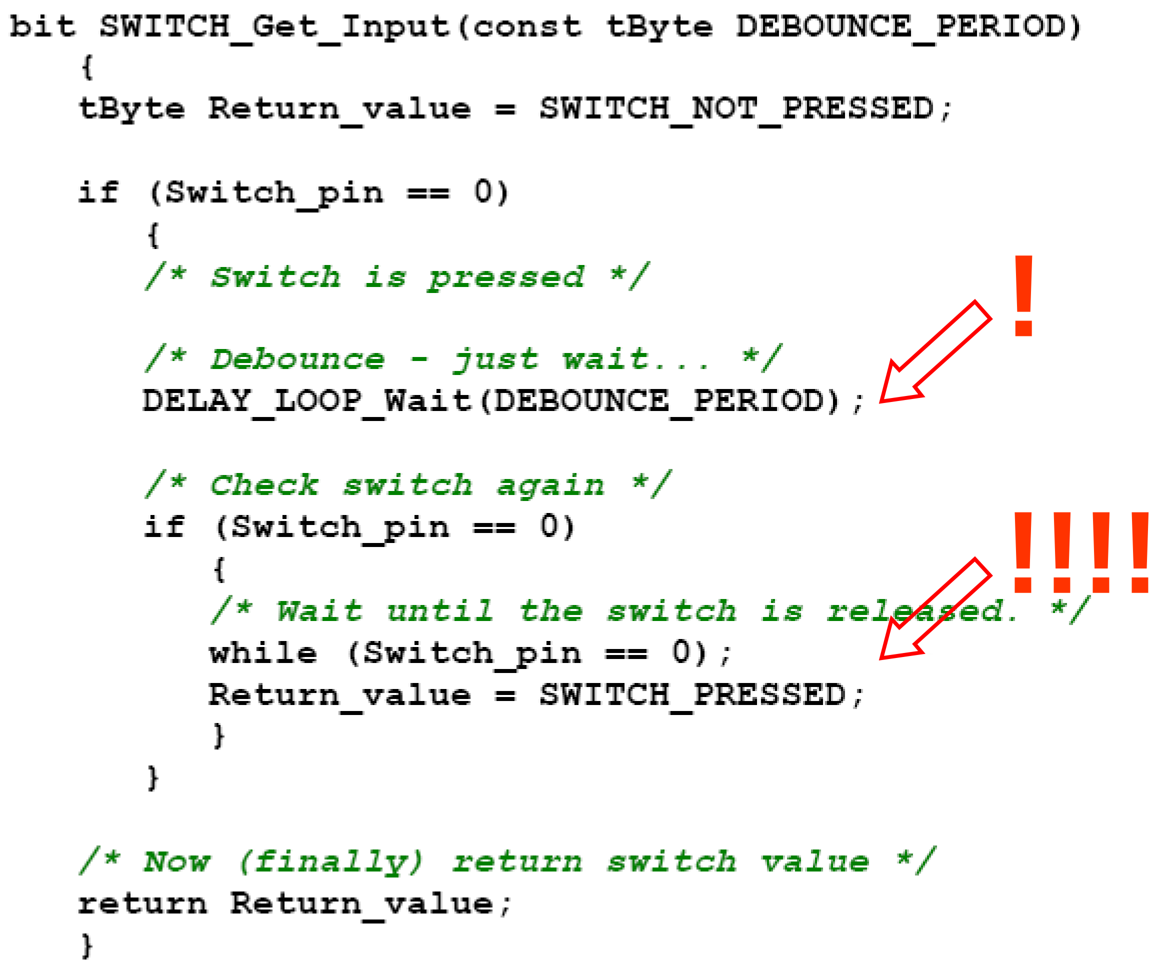


```
void main(void)
   {

   Init_System();

   while(1) /* 'for ever' (Super Loop) */
      {
      X();            /* Call the function (10 ms duration) */
      Delay_50ms(); /* Delay for 50 ms */
      }
   }
```

- Dobry sposób… pod warunkiem, że:
  - Znamy dokładny czas wykonywania funkcji X(),
  - Czas ten jest zawsze jednakowy… w praktyce ***nie realne!!!***

Michael J. Pont.

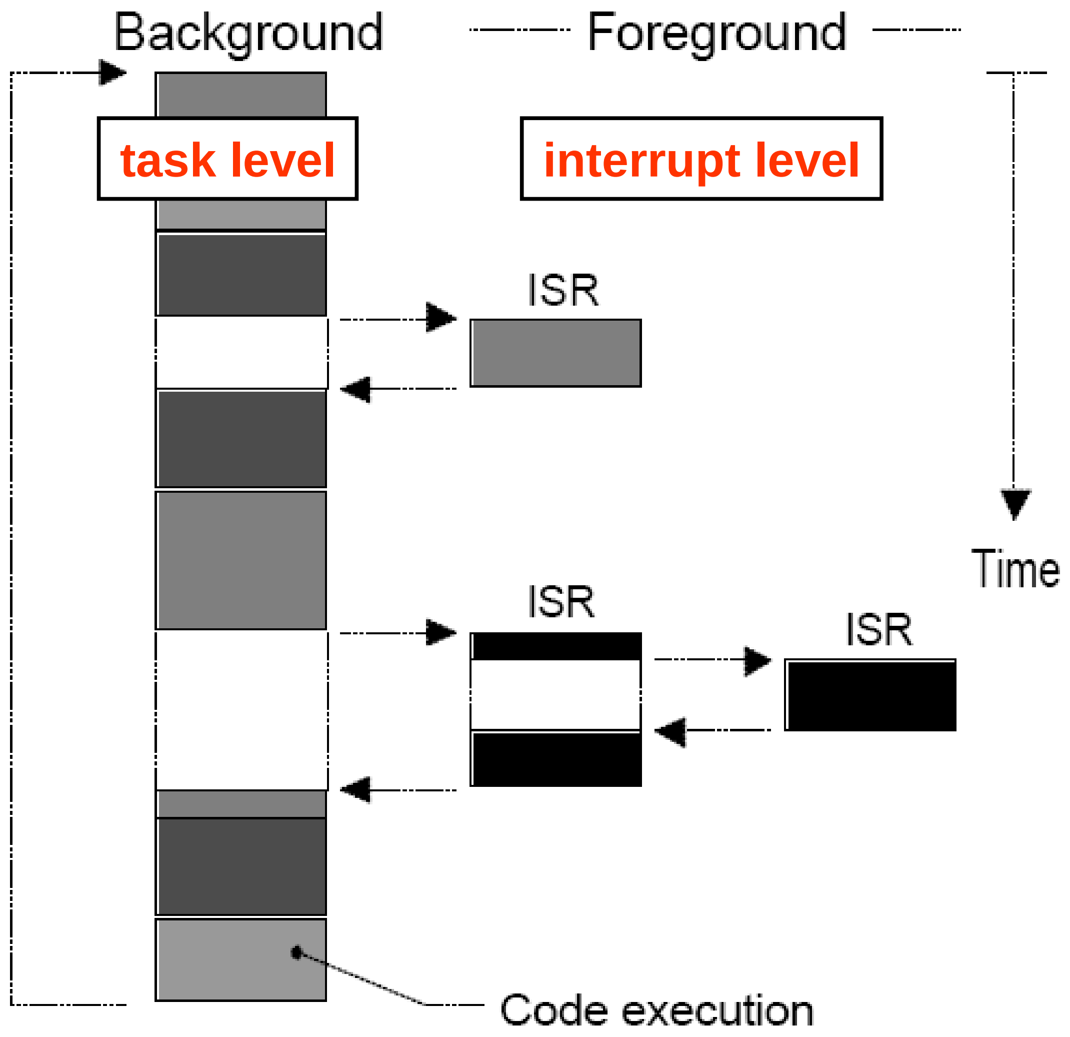
Super Loop: sposoby rozwiązania problemu…



```
bit SWITCH_Get_Input(const tByte DEBOUNCE_PERIOD)
   {
   tByte Return_value = SWITCH_NOT_PRESSED;

   if (Switch_pin == 0)
      {
      /* Switch is pressed */

      /* Debounce - just wait... */
      DELAY_LOOP_Wait(DEBOUNCE_PERIOD);

      /* Check switch again */
      if (Switch_pin == 0)
         {
         /* Wait until the switch is released. */
         while (Switch_pin == 0);
         Return_value = SWITCH_PRESSED;
         }
      }

   /* Now (finally) return switch value */
   return Return_value;
   }
```

- The execution time of typical code is not constant, it changes with successive passes through a portion of the loop,
- If a code change is made, the timing of the loop is affected,
- Most high volume microcontroller-based applications are designed as foreground/background systems.

Super Loop: sposoby rozwiązania problemu…



- Super Loop calls modules (functions) to perform desired operation
- ISRs handle asynchronous events and perform critical operations
- Worst case task level response time depends on how long the background loop takes to execute

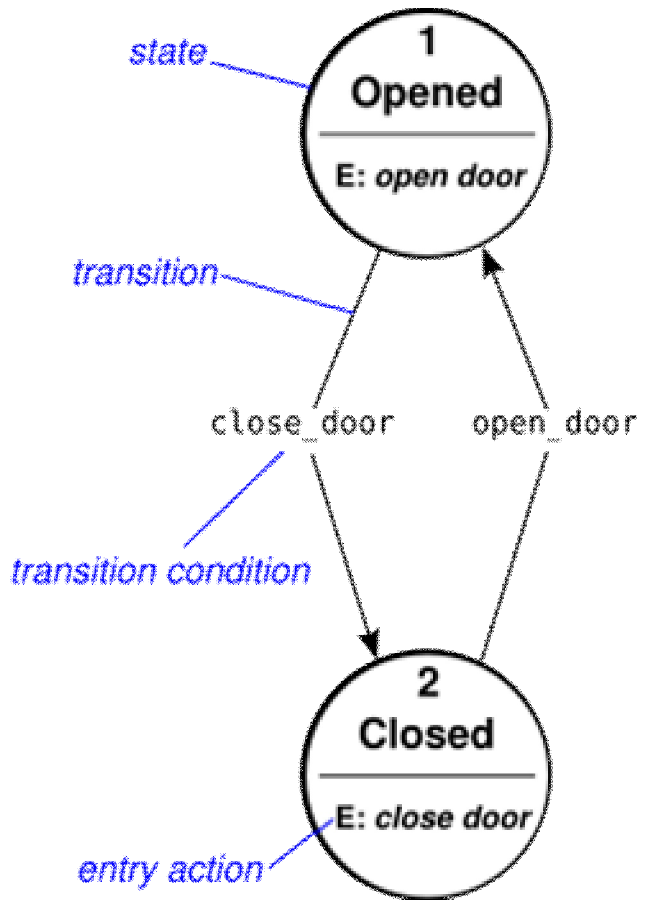
Co gdy program staje się bardziej skomplikowany?

- w pewnym momencie podeście typu „klasyczny Super Loop” przestaje wystarczać,
  - zadania krytyczne czasowo, realizowane na poziomie przerwań stają się bardziej rozbudowane (np. pojawia się zależność od kontekstu, stanu, w jakim jest urządzenie),
  - konieczność implementacji protokołów transmisji
  - komunikacja – np. modem GSM

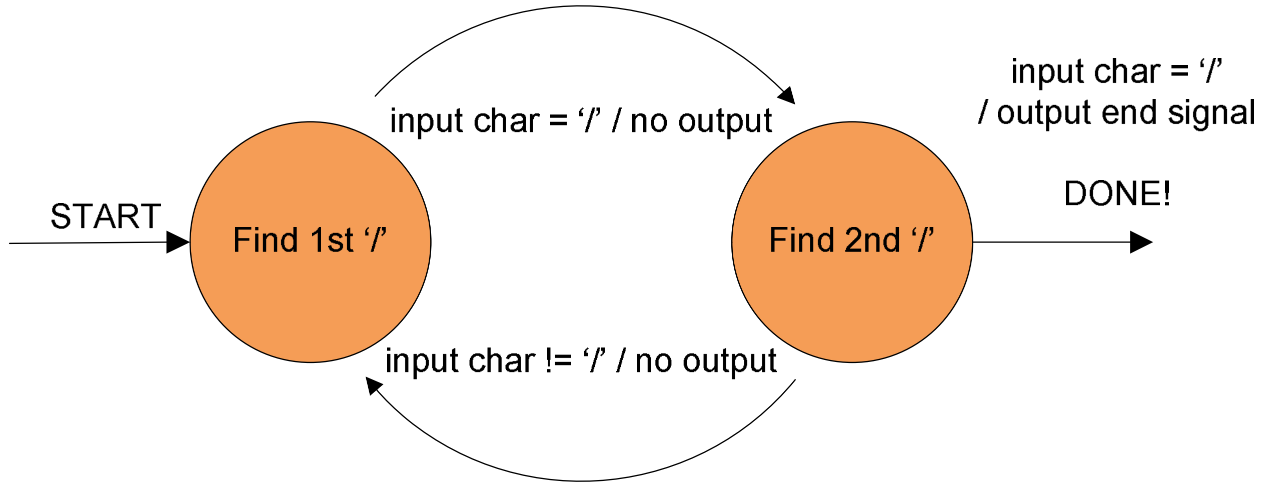
- najbardziej naturalnym podejściem staje się implementacja programu (jednego bądź większej ilości modułów) za pomocą maszyn stanu…

State Machines

- A state machine is defined as an algorithm that can be in one of a small number of states.
- A state is a condition that causes a prescribed relationship of inputs to outputs, and of inputs to next states.
- Mealy machine is a state machine where the outputs are a function of both present state and input. Moore machine, in which the outputs are a function only of state.
- In both cases, the next state is a function of both present state and input.



State Machines



A simple state machine to parse a character string, looking for '//'

- the first occurrence of a slash produces no output, but causes the machine to advance to the second state.
- if it encounters a non-slash while in the second state, then it will go back to the first state, because the two slashes must be adjacent.
- if it finds a second slash, however, then it produces the "we're done" output.

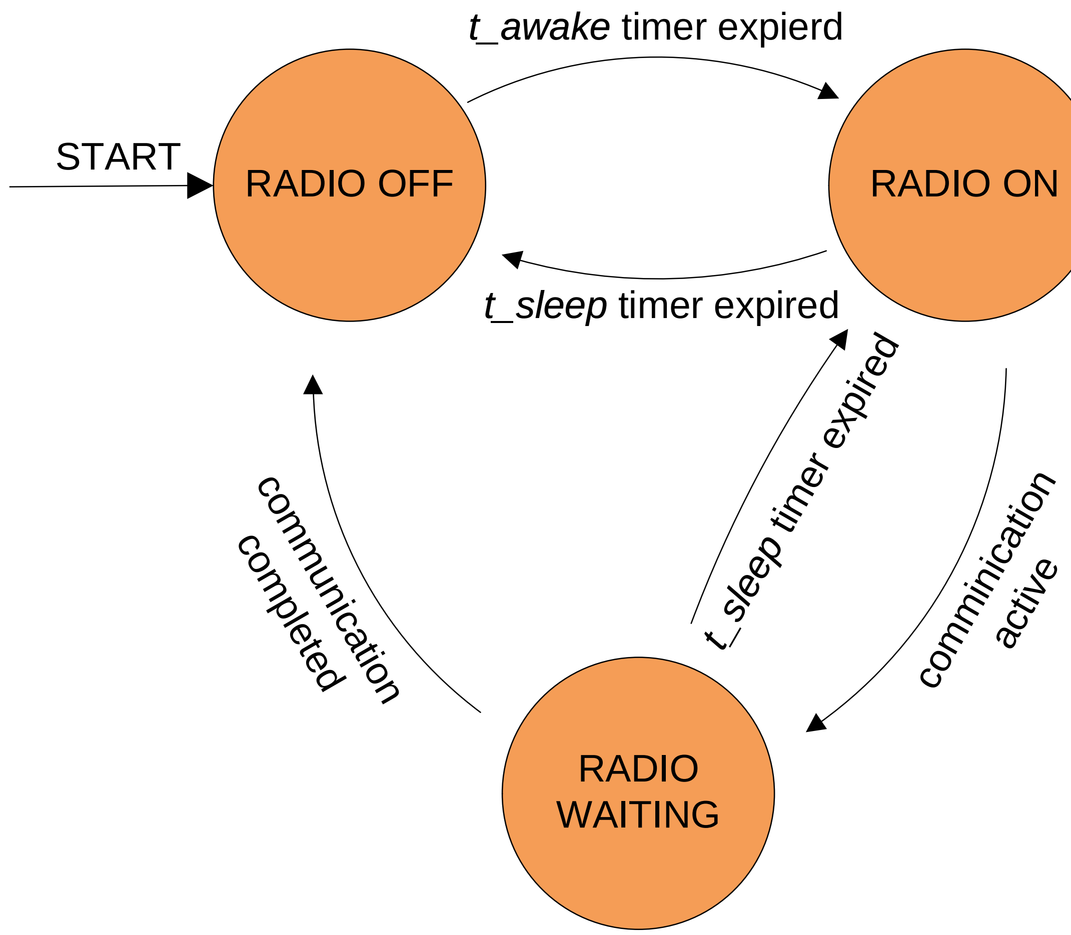
example of implementation - requirements

1. Turn radio on.
2. Wait for *t_awake* milliseconds.
3. Turn radio off, but only if all communication has completed.
4. If communication has not completed, wait until it has completed. Then turn off the radio.
5. Wait for *t_sleep* milliseconds. If the radio could not be turned off before *t_sleep* milliseconds because of remaining communication, do not turn the radio off at all.
6. Repeat from step 1.

Problem: with events, we can't write this as a 6-step program!

example of implementation – state machine

With events, we must use an explicit state machine!



example of implementation – C code (1)

```
enum {
  ON,
  WAITING,
  OFF
} state;
```

simple, but:

- it can lead to a very long function (for eg. 10 or 20 lines of code per state for each of 20 or 30 states)
- it can lead to astray when you change the code late in the testing phase (for eg. forgot a break statement at the end of a case)
- having one state's code "fall into" the next state's code is usually a no-no (can be used only when implicitly marked with for eg. `//fallthrough` comment)

```
void radio_wake_eventhandler() {
  switch(state) {
  case OFF:
    if(timer_expired(&timer)) {
       radio_on();
       state = ON;
       timer_set(&timer, T_AWAKE);
    }
    break;
  case ON:
    if(timer_expired(&timer)) {
       timer_set(&timer, T_SLEEP);
       if(!communication_complete()) {
          state = WAITING;
       } else {
          radio_off();
          state = OFF;
       }
    }
    break;
  case WAITING:
    if(communication_complete() || timer_expired(&timer)) {
       state = ON;
       timer_set(&timer, T_AWAKE);
    } else {
       radio_off();
       state = OFF;
    }
    break;
  }
}
```

switch statement, with a separate case for each state

example of implementation – C code (2)

```c
typedef enum {
  ON = 0,
  WAITING,
  OFF
} eState_t;

void (*state_table[])() = {RadioOn, RadioWait,
     RadioOff};
eState_t currentState;

main() {
  //...
  while (1) {
    decrementTimer();
  }
}

void radio_wake_eventhandler(void) {
  state_table[currentState]();
}
```

```c
void RadioOff(void) {
  if(timer_expired(&timer)) {
    radio_on();
    currentState = ON;
    timer_set(&timer, T_AWAKE);
  }
}

void RadioOn(void) {
  if(timer_expired(&timer)) {
    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {
      currentState = WAITING;
    } else {
      radio_off();
      currentState = OFF;
    }
  }
}

void RadioWait(void) {
  if(communication_complete() ||
                timer_expired(&timer)) {
    currentState= ON;
    timer_set(&timer, T_AWAKE);
  } else {
    radio_off();
    currentState = OFF;
  }
}
```

array of pointers to the individual state functions

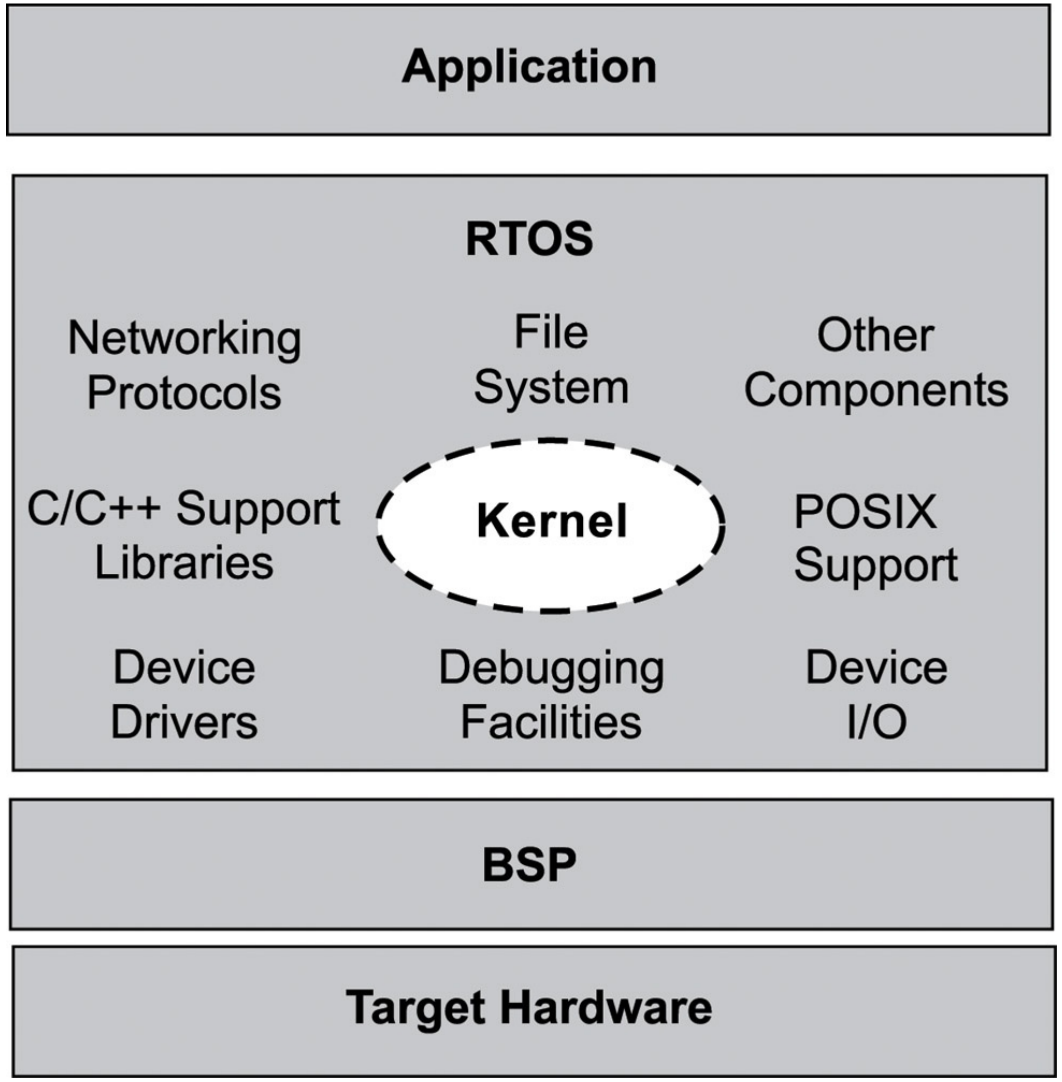
State Machines – sum up

- Advantages:
  - state machine can define various things: whole application, application module, protocol analyzer module, key debouncer etc.
  - state changes can be triggered by event-handler functions
  - forces the programmer to think of all the cases and, therefore, to extract all the required information from the user,
  - you can quickly draw a state transition diagram on a whiteboard, in front of the user, and walk him through it,
  - the test plan almost writes itself - all you have to do is to go through every state transition

Protothreads / Super Simple Tasker

- Advantages :
  - implements sequential flow of control without using complex state machines or full multi-threading
  - lightweight:
    - in traditional multi-threading may have a too large memory overhead for embedded system – for eg. each task requires its own stack,
    - in protothreads all threads run on the same stack, context switching is done by stack rewinding
  - requires only two bytes of memory per protothread,
  - implemented in pure C, do not require any machine-specific assembler code.

RTOS



- A real-time operating system (RTOS) is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code.
- Application code designed on an RTOS can be quite diverse, ranging from a simple application for a digital stopwatch to a much more complex application for aircraft navigation.
- Therefore a good RTOS is scalable in order to meet different sets of requirements for different applications.

In some applications, an RTOS comprises only a kernel, which is the core supervisory software that provides minimal logic, scheduling, and resource-management algorithms.

But RTOS can be a combination of various modules, including the kernel, a file system, networking protocol stacks, and other components required for a particular application

Kernel

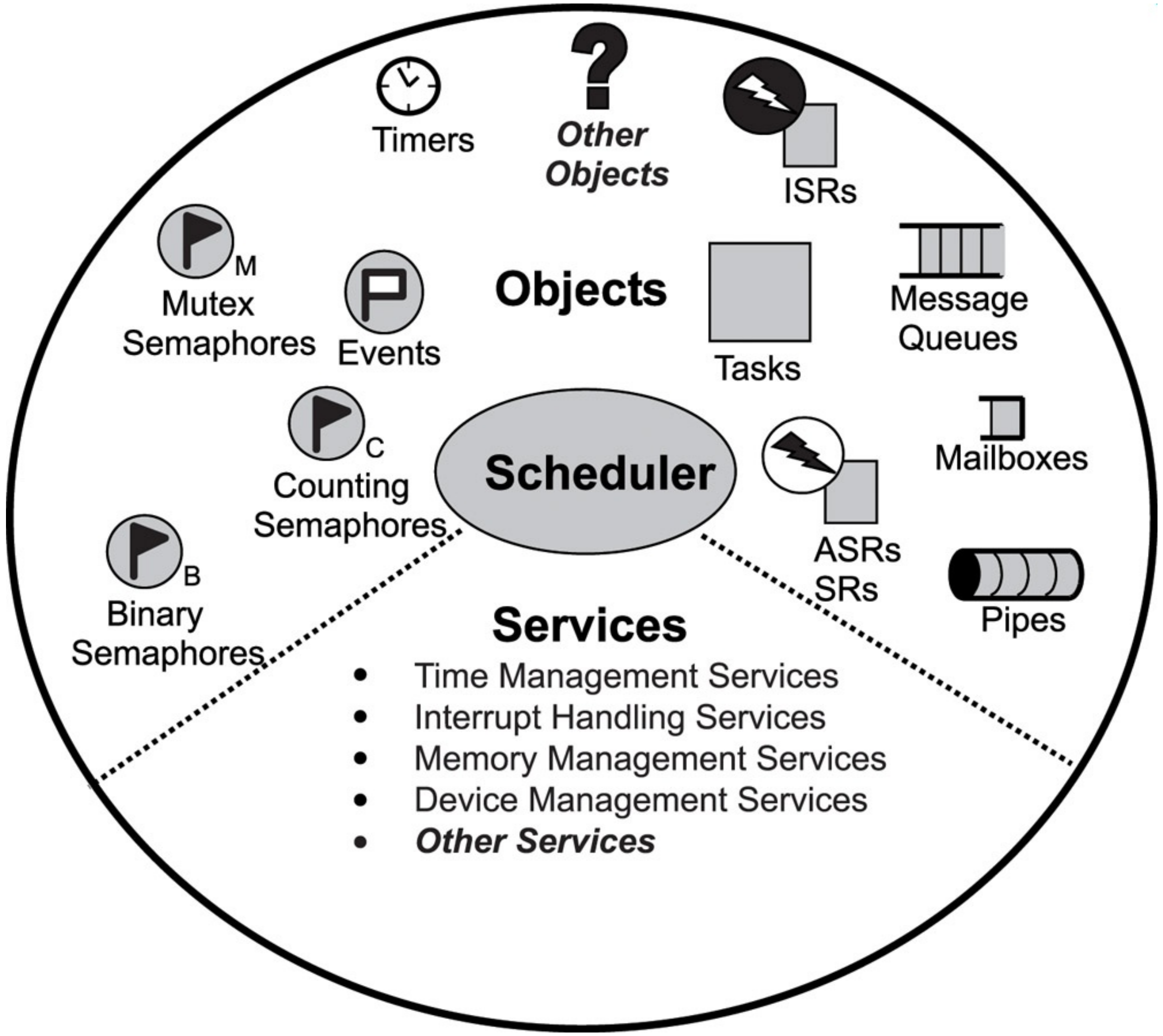
- ***Kernel*** – the part of a multitasking system responsible for the management of tasks (that is, for managing the CPU's time) and communication between tasks.

  The fundamental service provided by the kernel is context switching.

- The use of a real-time kernel will generally simplify the design of systems by allowing the application to be divided into multiple tasks managed by the kernel. A kernel will add overhead to your system and consume CPU time (typically between 2 and 5%).

- A kernel can allow you to make better use of your CPU by providing you with indispensible services such as semaphore management, mailboxes, queues, time delays, etc.

Kernel components



Common components in an RTOS kernel that including objects, the scheduler, and some services

- Scheduler – is contained within each kernel and follows a set of algorithms that determines which task executes when. Some common examples of scheduling algorithms include round-robin and preemptive scheduling.
- Objects – are special kernel constructs that help developers create applications for real-time embedded systems. Common kernel objects include tasks, semaphores, and message queues.
- Services – are operations that the kernel performs on an object or, generally operations such as timing, interrupt handling, and resource management.

Terminologia

- ***Task Level Response*** – time, between moments when information for a background module was made available by an ISR and when it was processed by the background routine.
- ***Critical Section Of Code, Critical Region -*** A critical section of code is code that needs to be treated indivisibly. Once the section of code starts executing, it must not be interrupted.
- ***Resource -*** A resource is any entity used by a task. A resource can thus be an I/O device such as a printer, a keyboard, a display, etc. or a variable, a structure, an array, etc
- ***Shared Resource -*** A shared resource is a resource that can be used by more than one task. Each task should gain exclusive access to the shared resource to prevent data corruption – this is called ***Mutual Exclusion.***

RTOS Overhead

- ***RTOS Overhead*** – code added by RTOS kernel. Kernel requires extra ROM (code space), additional RAM for its data structures but most importantly, each task requires its own stack space which has a tendency to eat up RAM quite quickly.

- RTOS overhead depends on
  - number of tasks,
  - how many registers the CPU has,
  - services provided by the kernel (communication between tasks etc.)

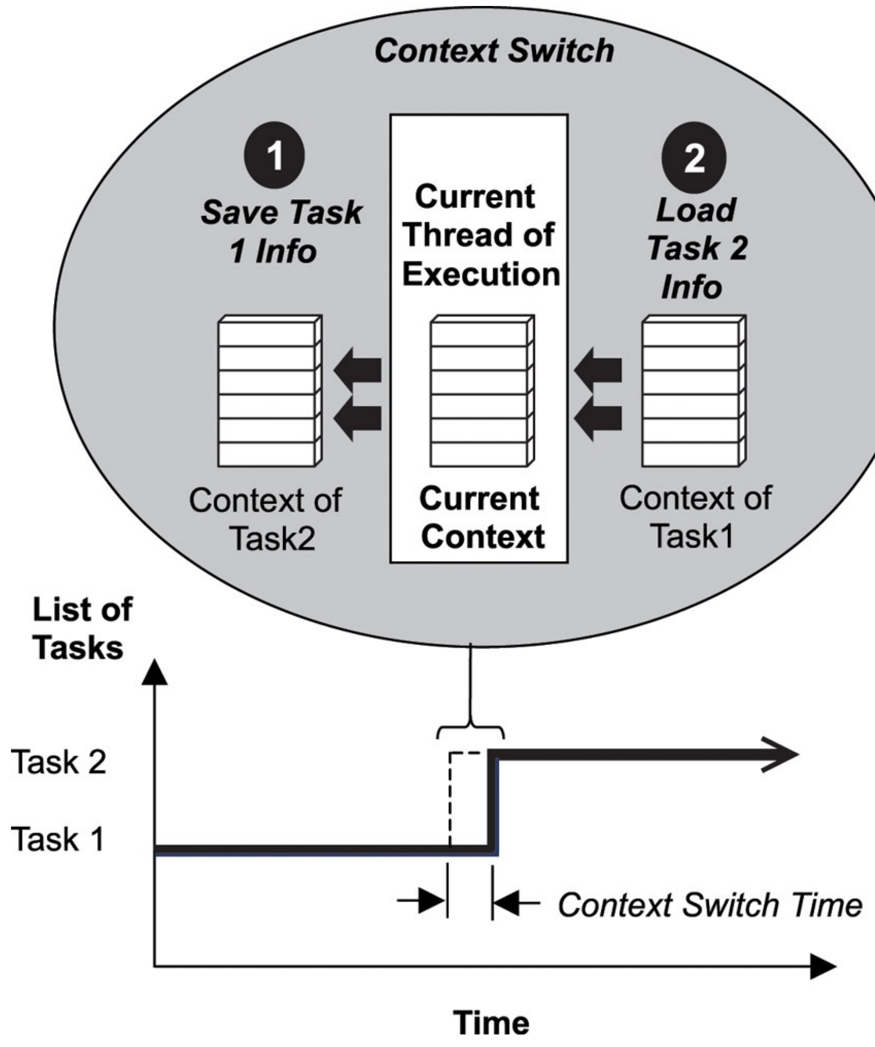
- The time required to perform a context switch (fundamental service provided by the kernel) is determined by how many registers have to be saved and restored by the CPU.
- Performance of a real-time kernel should not be judged on how many context switches the kernel is capable of doing per second.

Scheduler

- ***Scheduler*** – is the part of the kernel responsible for determining which task will run next.
- To understand how scheduling works one needs to know following terms:
  - schedulable entities
  - multitasking
  - context switching
  - dispatcher
  - scheduling algorithms

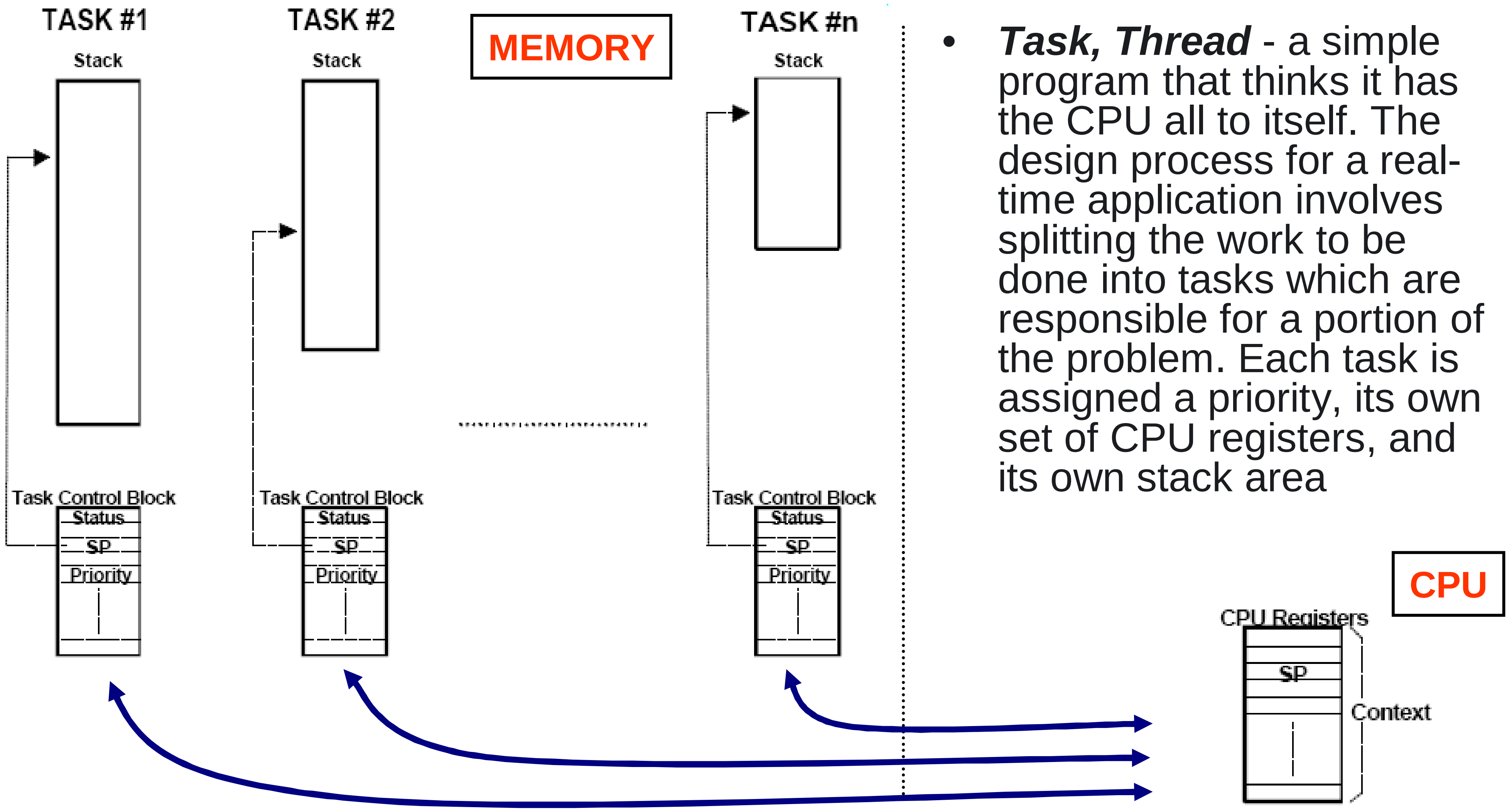
- ***Schedulable entities*** - schedulable entity is a kernel object that can compete for execution time on a system, based on a predefined scheduling algorithm; tasks and processes are all examples of schedulable entities found in most kernels,

Multitasking



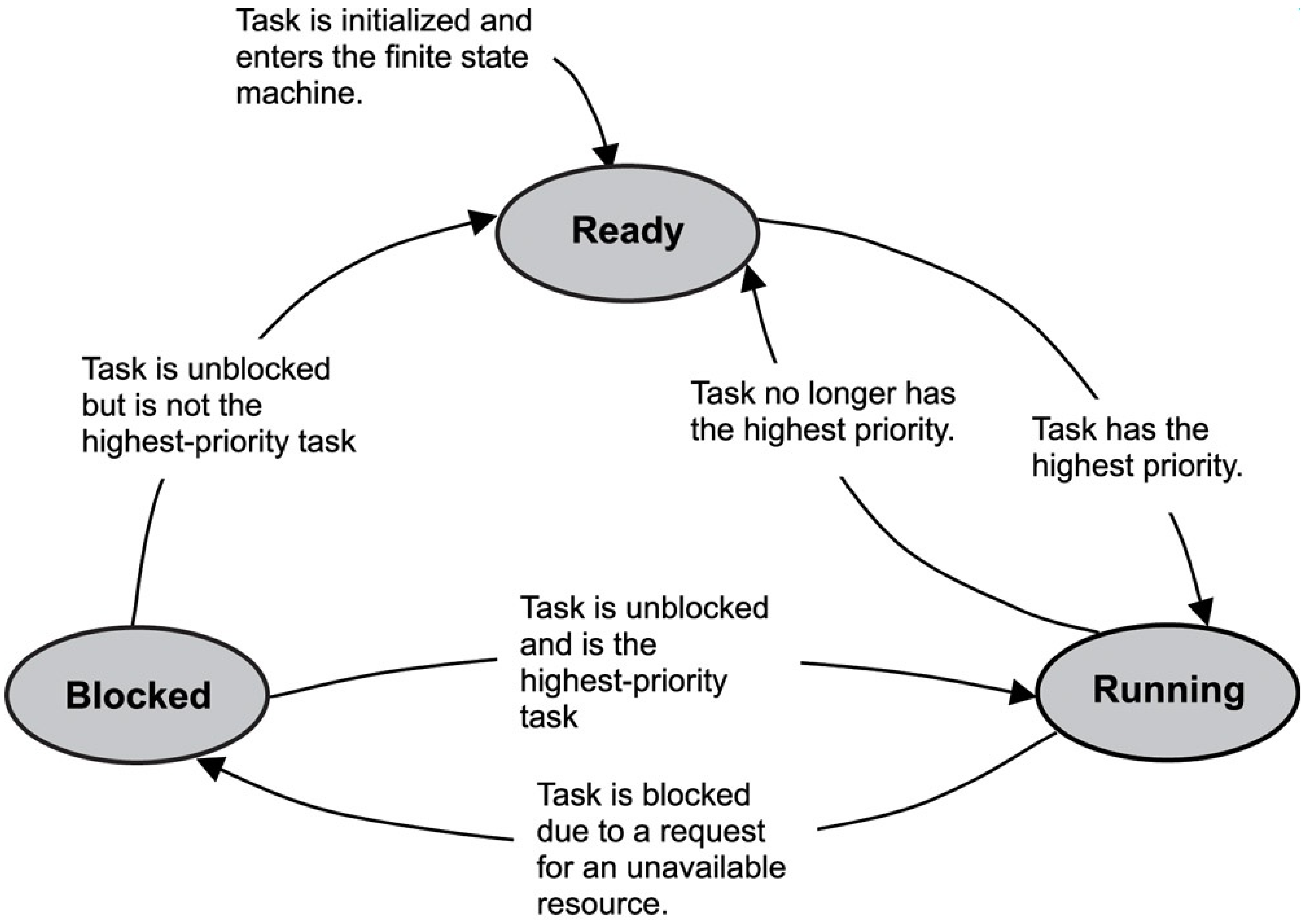
- ***Multitasking –*** Multitasking is the process of scheduling and switching the CPU between several tasks; a single CPU switches its attention between several sequential tasks;
- Multitasking is the ability of the operating system to handle multiple activities within set deadlines. A real-time kernel might have multiple tasks that it has to schedule to run.
- Multitasking maximizes the utilization of the CPU and also provides for modular construction of applications.
- One of the most important aspects of multitasking is that it allows the application programmer to manage complexity inherent in real-time applications. Application programs are typically easier to design and maintain if multitasking is used.

Task, Thread



- ***Task, Thread*** - a simple program that thinks it has the CPU all to itself. The design process for a real-time application involves splitting the work to be done into tasks which are responsible for a portion of the problem. Each task is assigned a priority, its own set of CPU registers, and its own stack area

Task States


Task is initialized and enters the finite state machine.
Ready
Task is unblocked but is not the highest-priority task
Task no longer has the highest priority.
Task has the highest priority.
Task is unblocked and is the highest-priority task
Blocked
Running
Task is blocked due to a request for an unavailable resource.

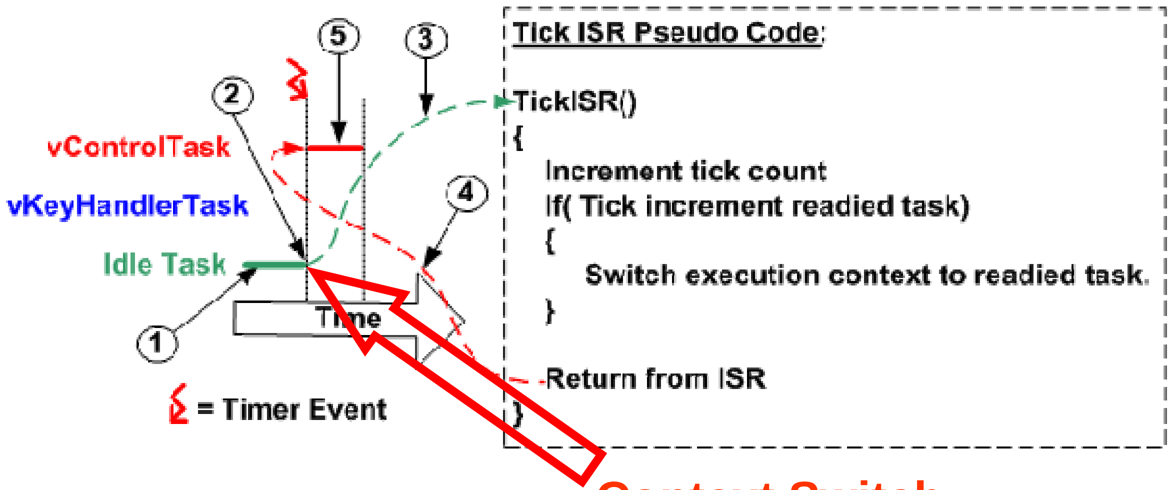
Context Switch / Task Switch

- ***Context Switch / Task Switch*** – When a multitasking kernel (it’s scheduler) decides to run a different task, it simply saves the current *task's context* (CPU registers) in the current task's context storage area – it’s stack. Once this operation is performed, the new task's context is restored from its storage area and then resumes execution of the new task's code.

Dispatcher

- The ***dispatcher*** is the part of the scheduler that performs context switching and changes the flow of execution. At any time an RTOS is running, the flow of execution, also known as flow of control, is passing through one of three areas: through an application task, through an ISR, or through the kernel.
- When a task or ISR makes a system call, the flow of control passes to the kernel to execute one of the system routines provided by the kernel. When it is time to leave the kernel, the dispatcher is responsible for passing control to one of the tasks in the user’s application. It will not necessarily be the same task that made the system call.

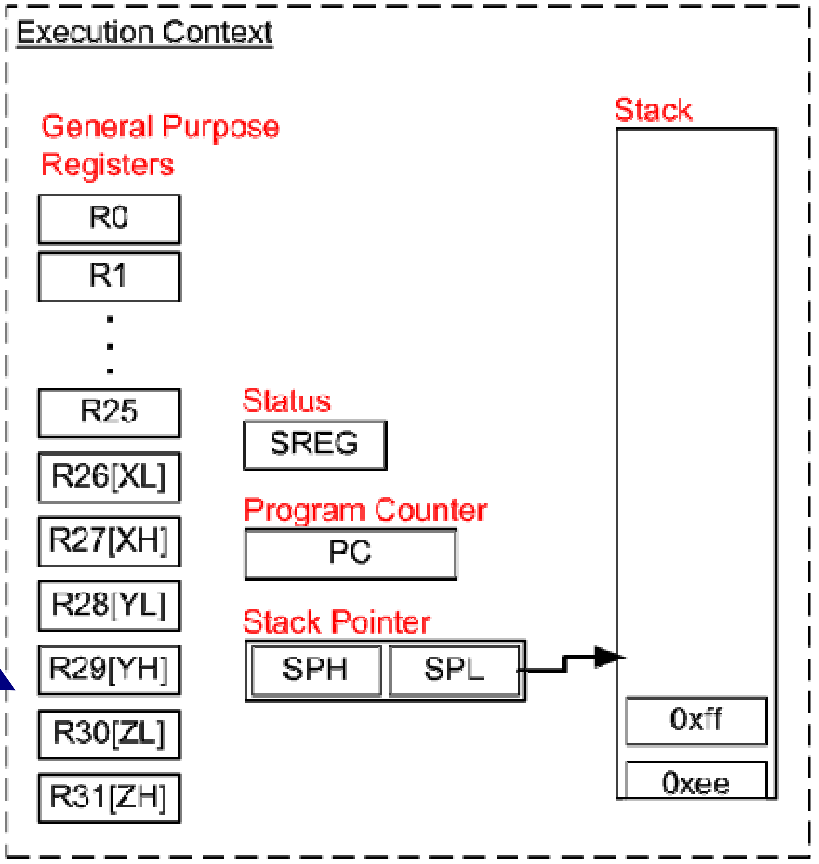
Context Switch on an AVR



```
Tick ISR Pseudo Code:

TickISR()
{
    Increment tick count
    If( Tick increment readied task)
    {
        Switch execution context to readied task.
    }

    Return from ISR
}
```

Context Switch

Current Execution Context will be switched - vControlTask context is to be loaded



what can cause context switch:
- RTOS kernel (it will suspend and resume tasks as necessary to ensure the task with highest priority that is ready to run is the task given processing time,
- task can choose to suspend itself:
  - sleep for a fixed period of time (wait a timeout),
  - wait for a resource to become available

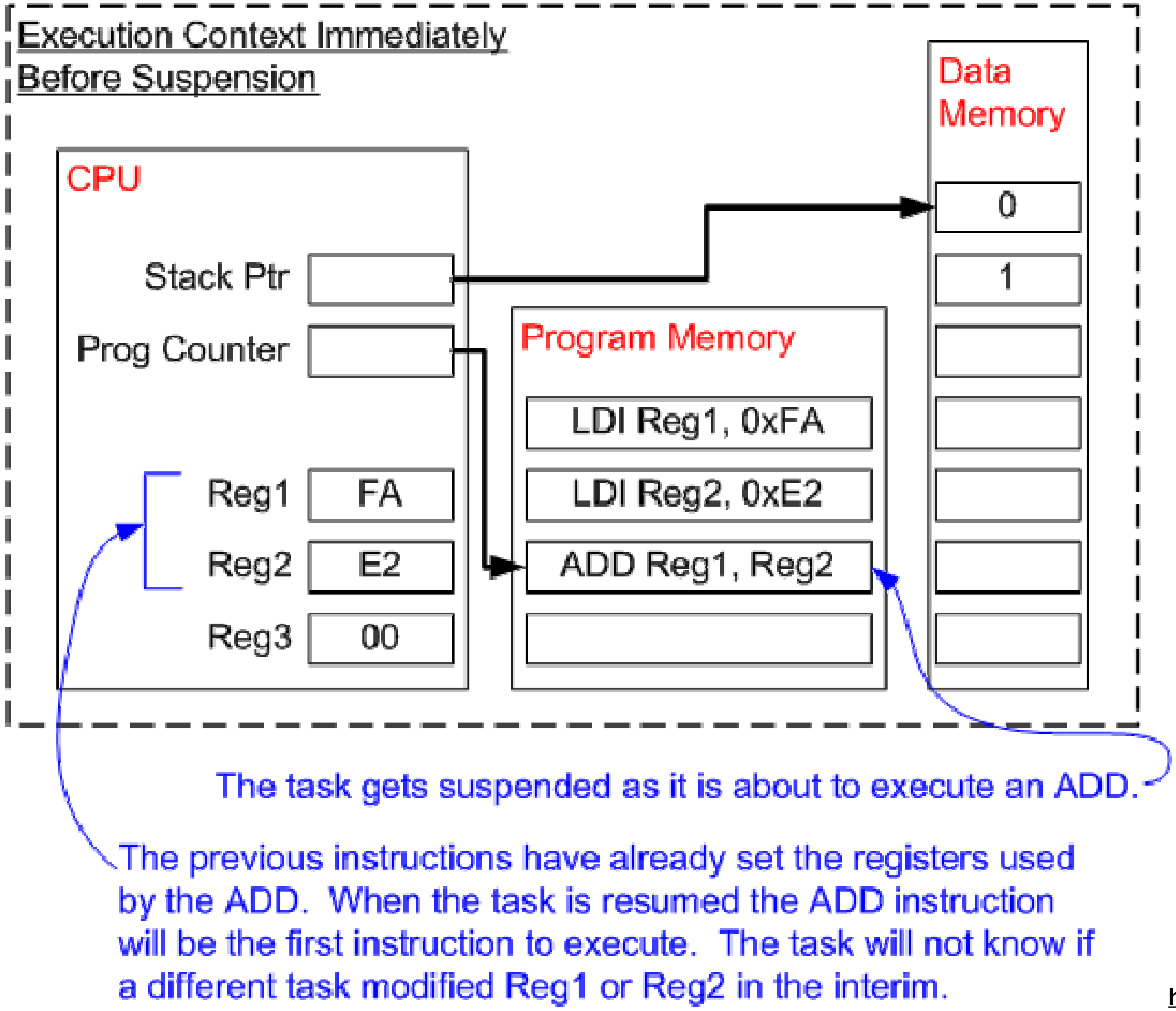
http://masters.donntu.edu.ua/2006/fvti/taitsky/library/art5.htm

Context Switch on an AVR

```
void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal, naked ) );
void vPortYieldFromTick( void ) __attribute__ ( ( naked ) );

void SIG_OUTPUT_COMPARE1A( void ) {     // Interrupt service routine for the RTOS tick.
  // Call the tick function.
  vPortYieldFromTick();
  // Return from the interrupt.  If a context switch has occurred this
  // will return to a different task.
  asm volatile ( "reti" );
}
void vPortYieldFromTick(void ) {
  // This is a naked function so the context is saved.
  portSAVE_CONTEXT();
  // Increment the tick count and check to see if the new tick value has caused a
  // delay period to expire. This function call can cause a task to become ready to run.
  vTaskIncrementTick();
  // See if a context switch is required. Switch to the context of a task made ready
  // to run by vTaskIncrementTick() if it has a priority higher than the interrupted task.
  vTaskSwitchContext();
  // Restore the context. If a context switch has occurred this will restore
  // the context of the task being resumed.
  portRESTORE_CONTEXT();
  // Return from this naked function.
  asm volatile ( "ret" );
}
```

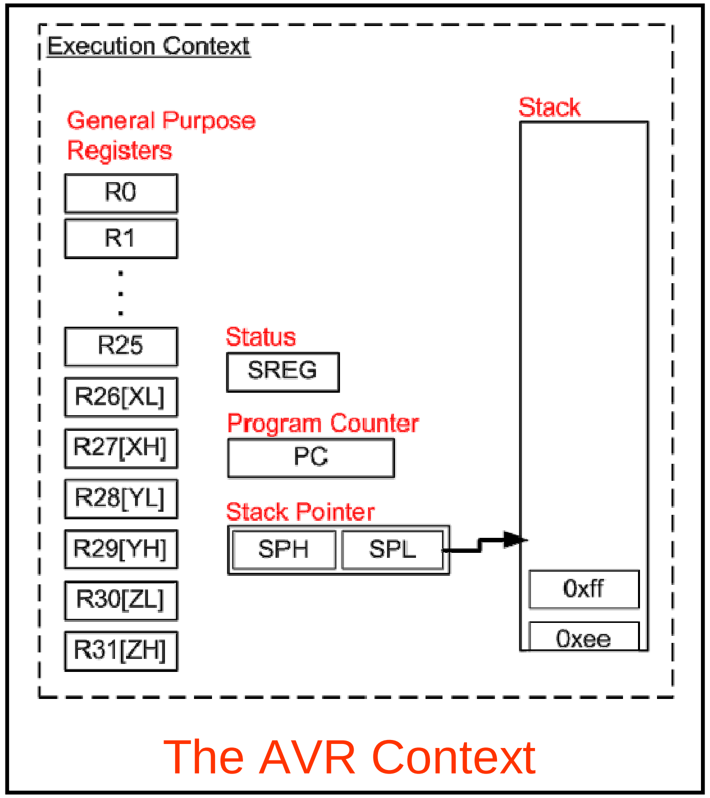
http://masters.donntu.edu.ua/2006/fvti/taitsky/library/art5.htm

Context Switch on an AVR



The task gets suspended as it is about to execute an ADD.

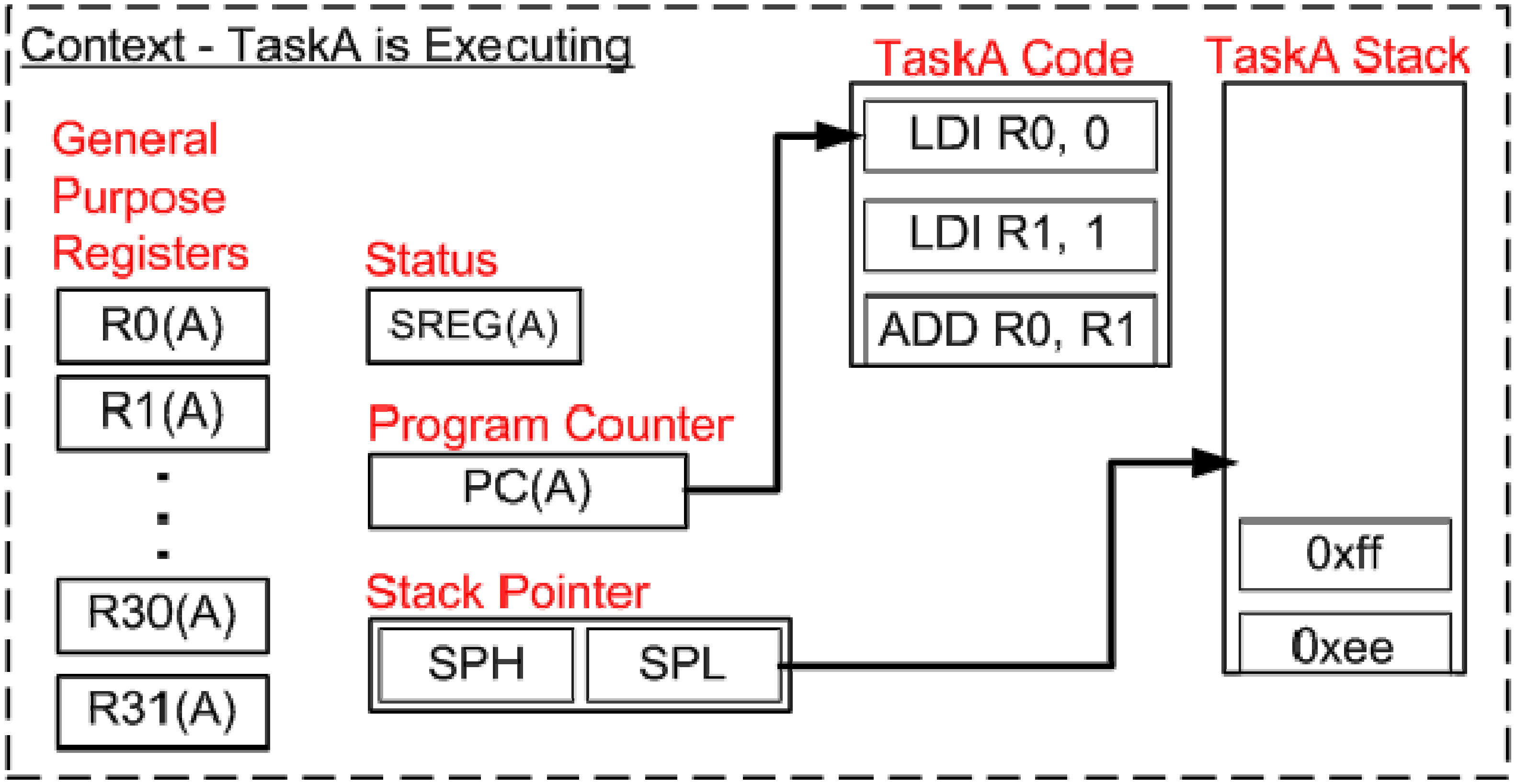
The previous instructions have already set the registers used by the ADD. When the task is resumed the ADD instruction will be the first instruction to execute. The task will not know if a different task modified Reg1 or Reg2 in the interim.



The AVR Context

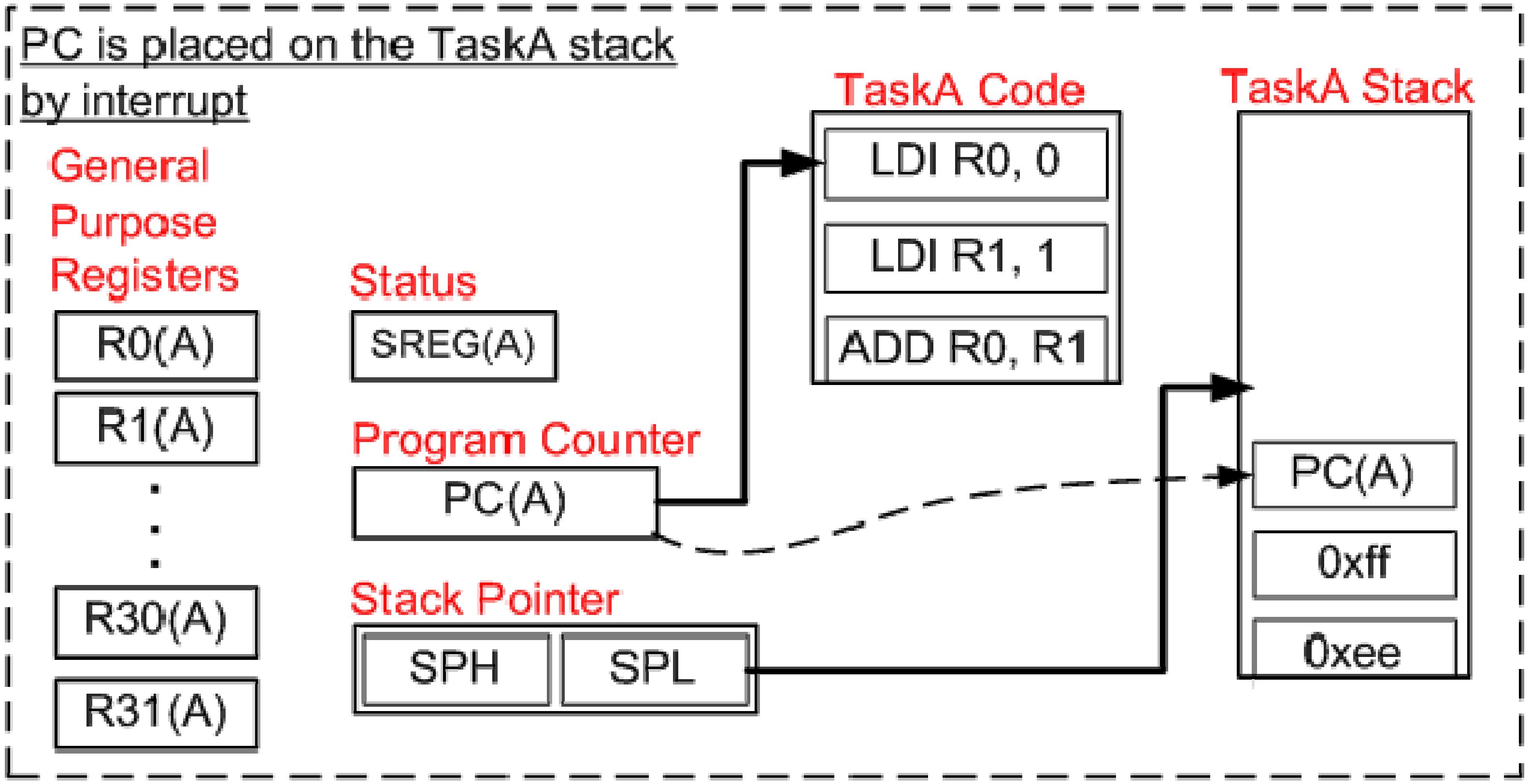
http://masters.donntu.edu.ua/2006/fvti/taitsky/library/art5.htm

Context Switch Step 1: Prior to the RTOS tick interrupt



http://masters.donntu.edu.ua/2006/fvti/taitsky/library/art5.htm

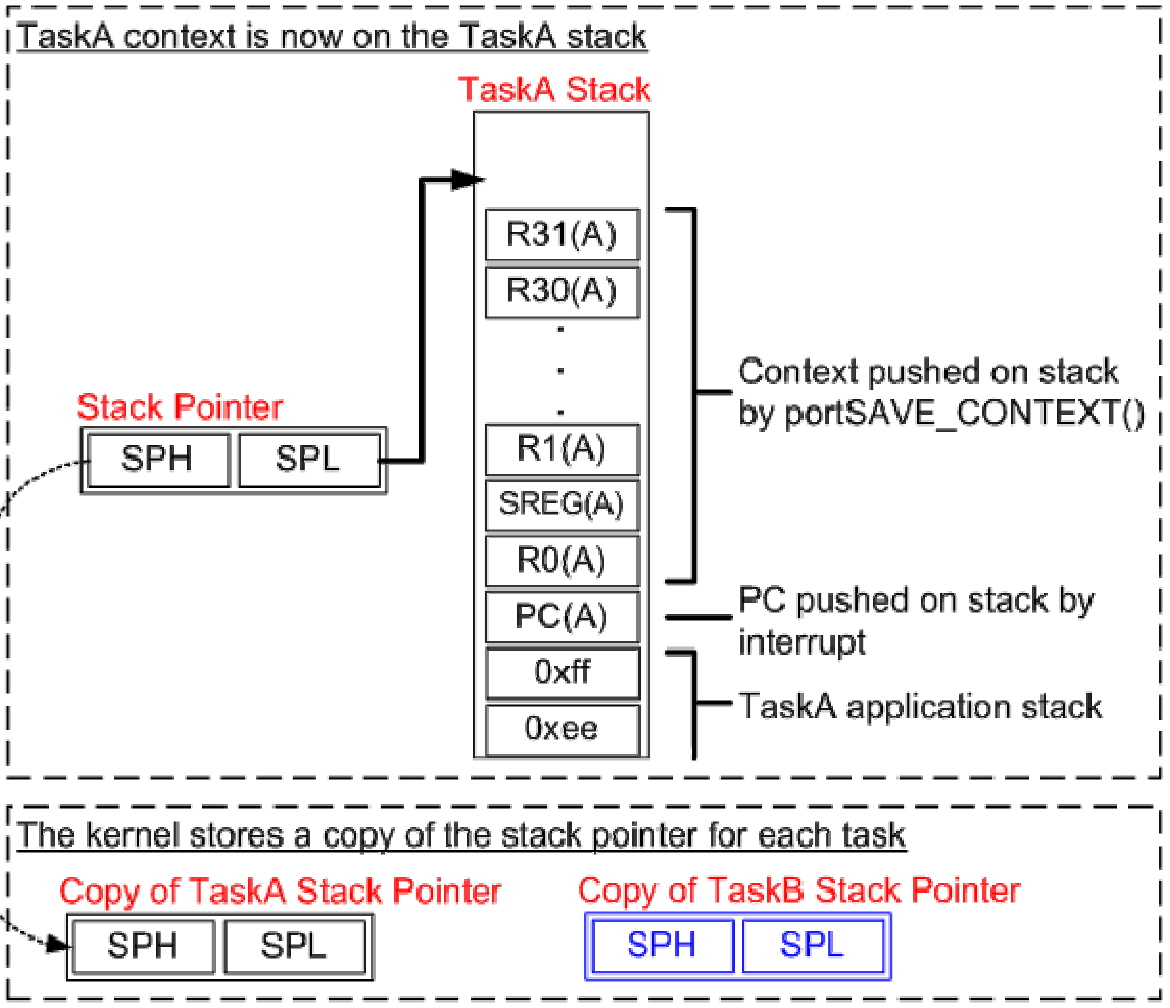
Context Switch Step 2: The RTOS tick interrupt occurs



http://masters.donntu.edu.ua/2006/fvti/taitsky/library/art5.htm

Context Switch Step 3: The RTOS tick interrupt executes

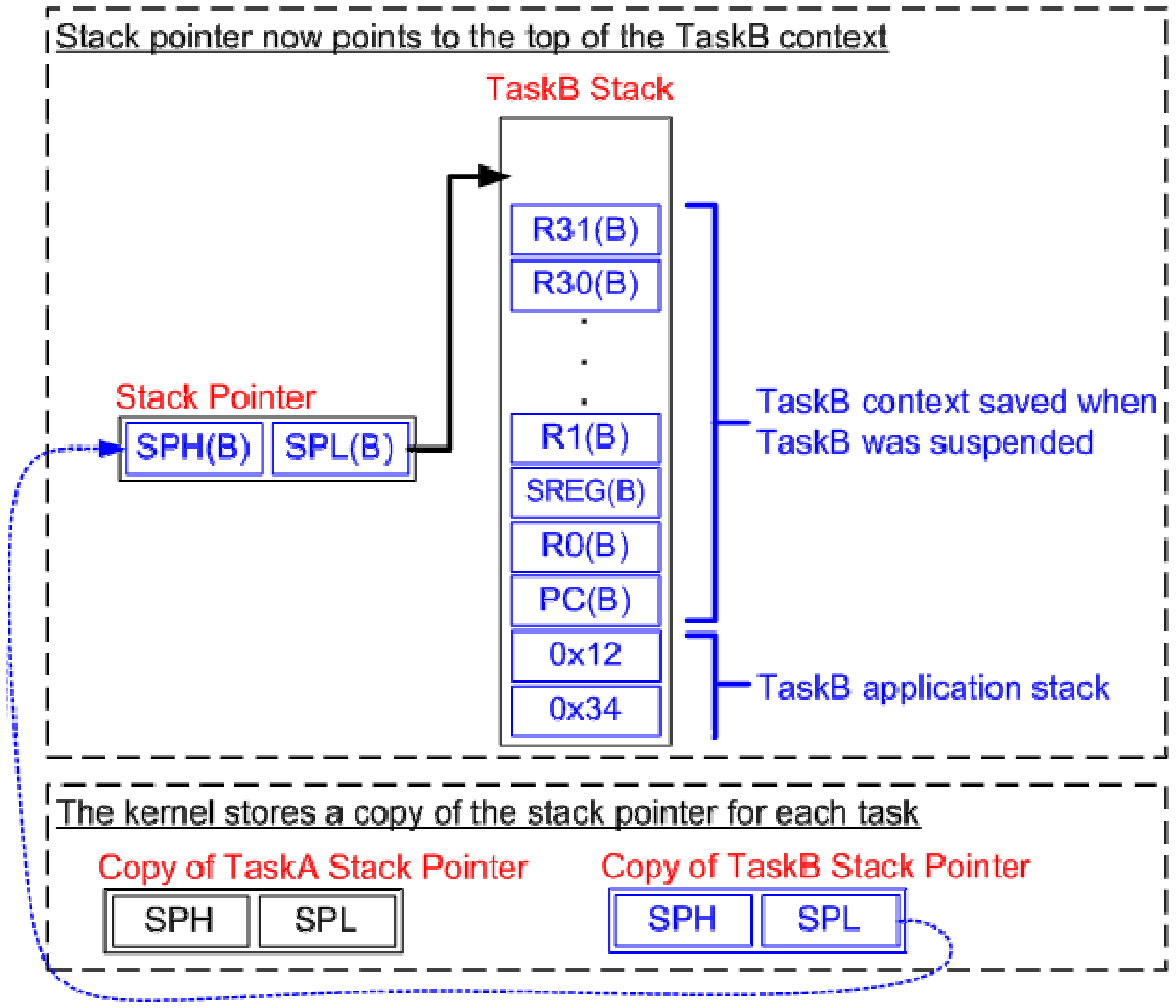
Entire AVR execution context is pushed onto the stack of TaskA


TaskA context is now on the TaskA stack
TaskA Stack
Stack Pointer
SPH
SPL
R31(A)
R30(A)
R1(A)
SREG(A)
R0(A)
PC(A)
0xff
0xee
Context pushed on stack by portSAVE_CONTEXT()
PC pushed on stack by interrupt
TaskA application stack
The kernel stores a copy of the stack pointer for each task
Copy of TaskA Stack Pointer
SPH
SPL
Copy of TaskB Stack Pointer
SPH
SPL

Context Switch Step 4: Incrementing the Tick Count

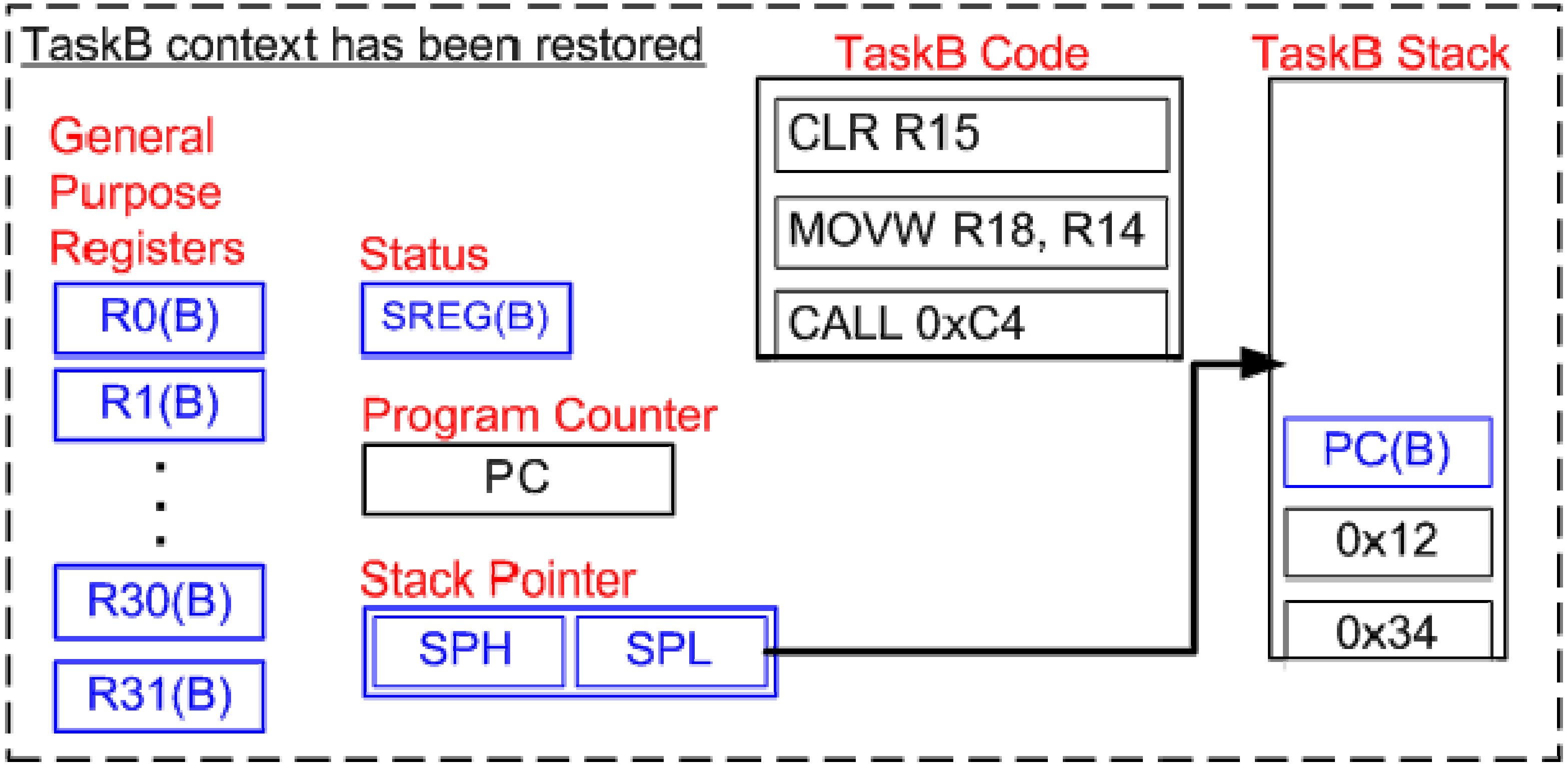
We assume that incrementing the tick count has caused TaskB to become ready to run. TaskB has a higher priority than TaskA so vTaskSwitchContext() selects TaskB as the task to be given processing time when the ISR completes

Context Switch Step 5: The TaskB stack pointer is retrieved


Stack pointer now points to the top of the TaskB context
TaskB Stack
Stack Pointer
SPH(B)
SPL(B)
R31(B)
R30(B)
R1(B)
SREG(B)
R0(B)
PC(B)
0x12
0x34
TaskB context saved when TaskB was suspended
TaskB application stack
The kernel stores a copy of the stack pointer for each task
Copy of TaskA Stack Pointer
SPH
SPL
Copy of TaskB Stack Pointer
SPH
SPL

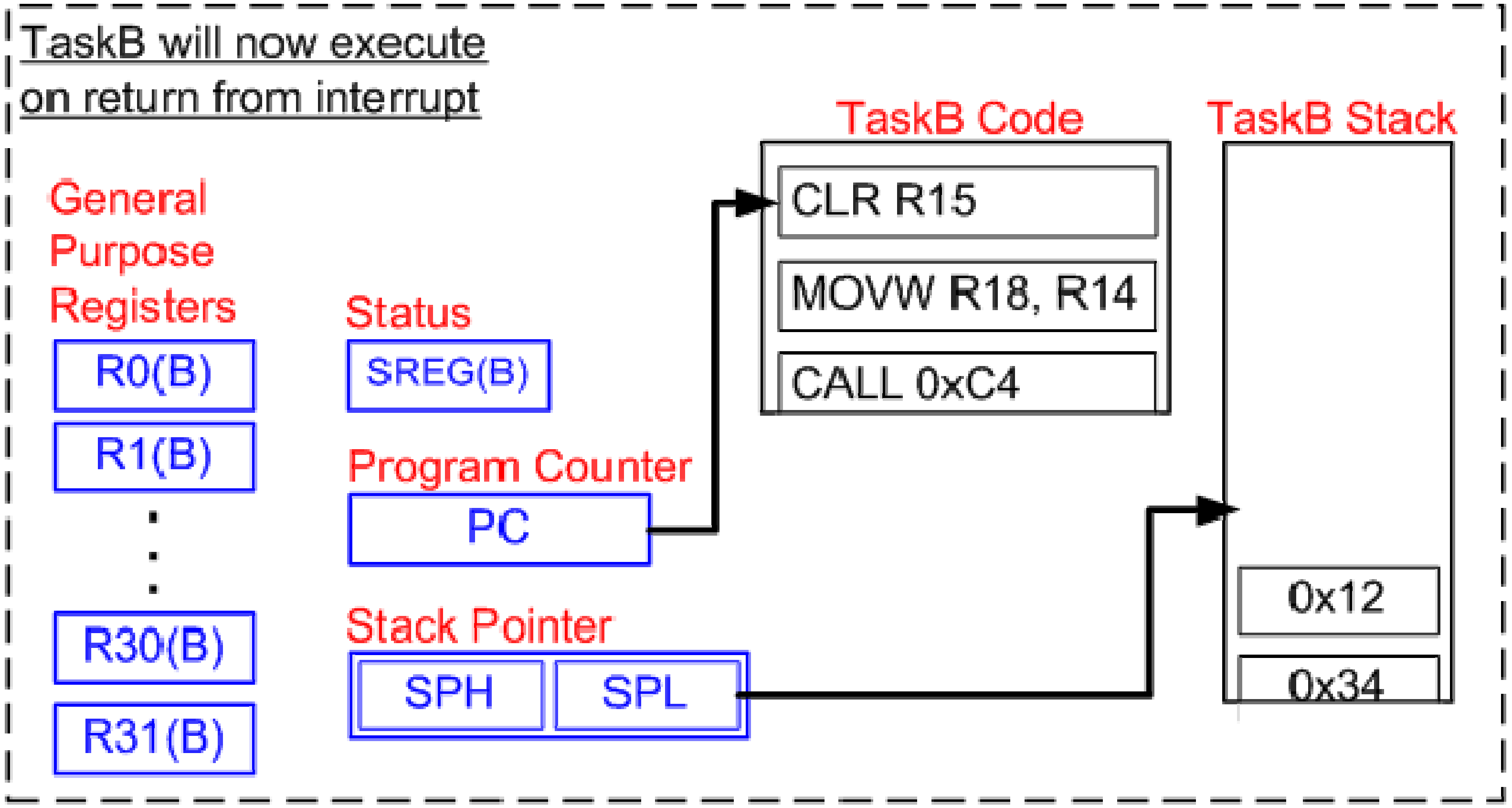
http://masters.donntu.edu.ua/2006/fvti/taitsky/library/art5.htm

Context Switch Step 6: Restore the TaskB context



The RTOS tick interrupt interrupted **TaskA**, but is returning to **TaskB** - the context switch is complete!

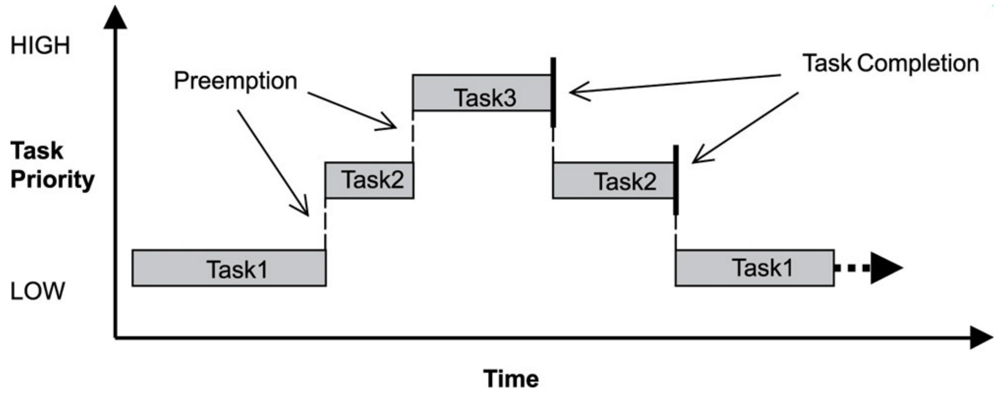
Context Switch Step 7: The RTOS tick exits



Scheduling Algorithms

- *The scheduler* determines which task runs by following a scheduling algorithm (also known as *scheduling policy*). Most kernels today support two common scheduling algorithms:
  - – preemptive priority-based scheduling
  - – non-preemptive priority-based scheduling

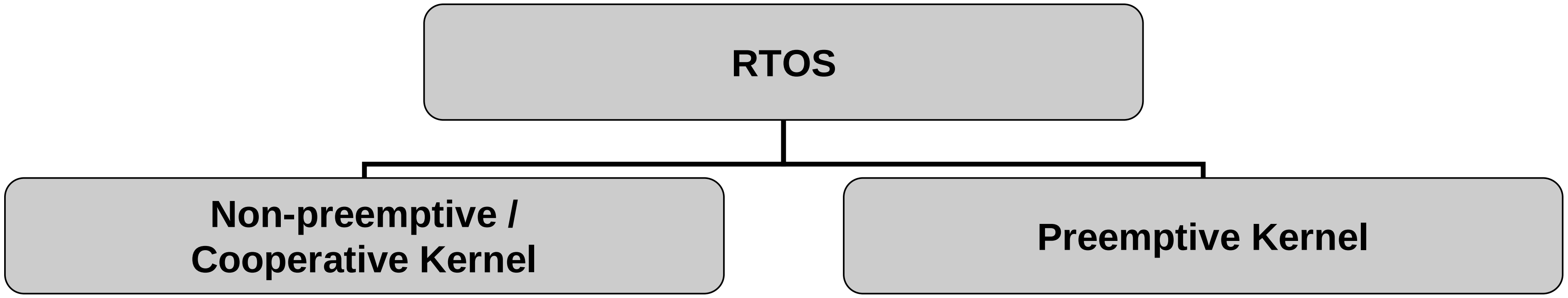
Example of preemptive priority-based scheduling



Scheduling Algorithms

- Most real-time kernels are priority based. Each task is assigned a priority based on its importance. The priority for each task is application specific. In a priority-based kernel, control of the CPU will always be given to the highest priority task ready-to-run. When the highest-priority task gets the CPU, however, is determined by the type of kernel used.
- There are two types of priority-based kernels: *non-preemptive* and *preemptive*

Types of priority-based kernels



Non-preemptive / Cooperative Kernel

- Each task does something to explicitly give up control of the CPU. To maintain the illusion of concurrency, this process must be done frequently.
- Asynchronous events are handled by ISRs.
- ISR can make a higher priority task ready to run, but the ISR always returns to the interrupted task.
- The new higher priority task will gain control of the CPU only when the current task gives up the CPU.

Preemptive Kernel

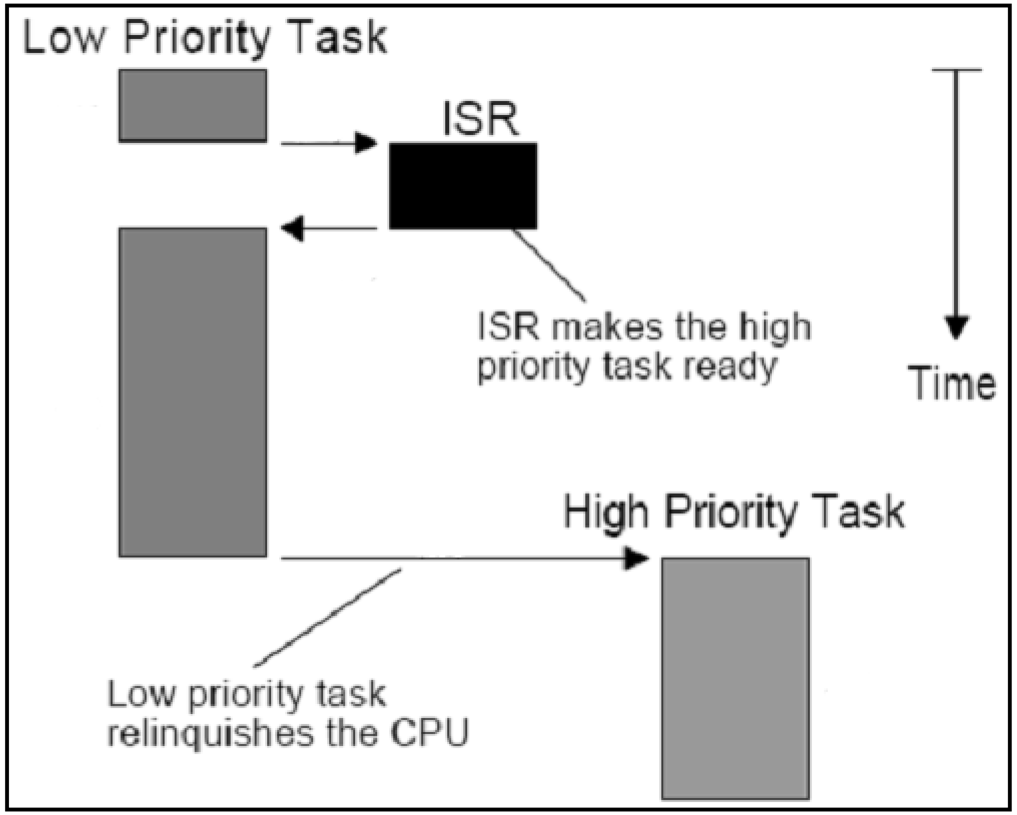
- Highest priority task ready to run is always given control of the CPU
- When a task makes a higher priority task ready to run, the current task is preempted (suspended) and the higher priority task is immediately given control of the CPU
- If an ISR makes a higher priority task ready, when the ISR completes, the interrupted task is suspended and the new higher priority task is resumed
- Execution of the highest priority task is deterministic.

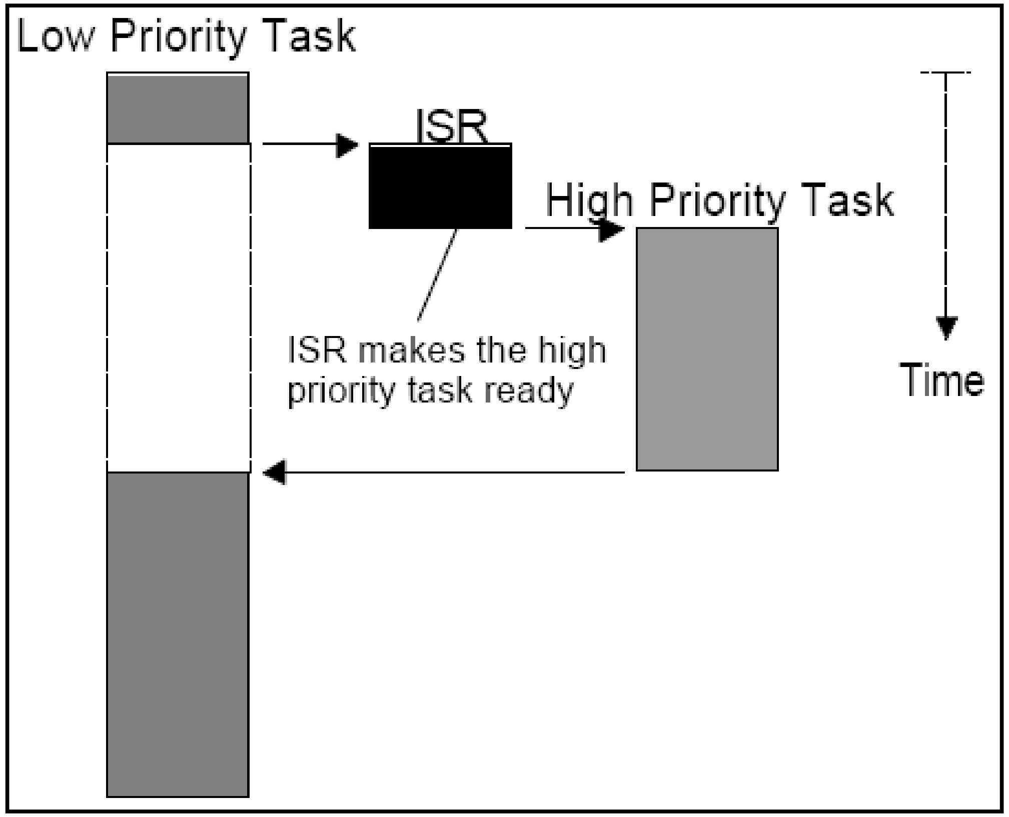
Types of priority-based kernels

RTOS

Non-preemptive / Cooperative Kernel

Preemptive Kernel


Low Priority Task
ISR
ISR makes the high priority task ready
Time
High Priority Task
Low priority task relinquishes the CPU



Non-preemptive / Cooperative Kernel

- Zalety
  - Interrupt latency is typically low
  - Non-reentrant functions can be used by each task without fear of corruption by another task, however, they should not be allowed to give up control of the CPU
  - Task-level response using a non-preemptive kernel can be much lower than with foreground/background systems
  - Lesser need to guard shared data through the use of semaphores – each task owns the CPU and you don't have to fear that a task will be preempted (but in some instances, semaphores should still be used – shared I/O devices may still require the use of mutual exclusion semaphores)
- Wady
  - Responsiveness - A higher priority task that has been made ready to run may have to wait a long time to run, because the current task must give up the CPU when it is ready to do so
  - Task-level response time is non-deterministic
  - You never really know when the highest priority task will get control of the CPU

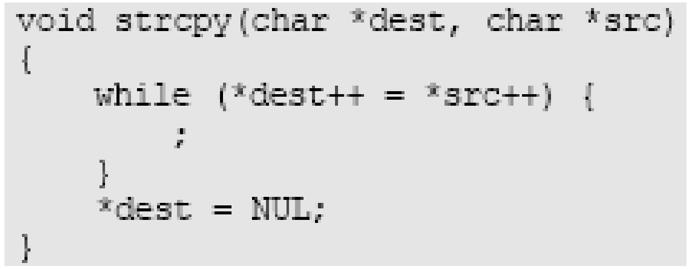
Preemptive Kernel

- Zalety
  - Great responsiveness - the highest priority task ready to run is always given control of the CPU.
  - Execution of the highest priority task is deterministic.
  - Task-level response time is minimized.

- Wady
  - Larger RTOS overhead
  - Application code using a preemptive kernel should not make use of non-reentrant functions unless exclusive access to these functions is ensured through the use of mu tual exclusion semaphores, because both a low priority task and a high priority task can make use of a common function. Corruption of data may occur if the higher priority task preempts a lower priority task that is making use of the function.

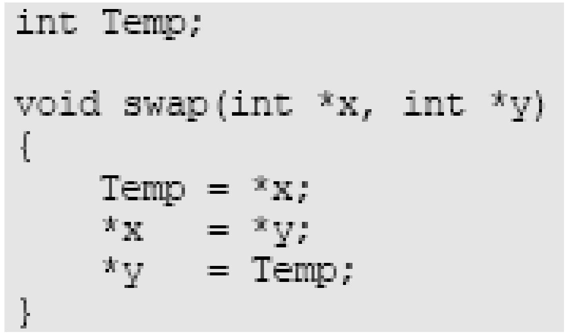
Reentrant Function

- Function that can be used by more than one task without fear of data corruption (function can be interrupted at any time and resumed at a later time without loss of data). Reentrant functions either use local variables (i.e., CPU registers or variables on the stack) or protect data when global variables are used.



```
void strcpy(char *dest, char *src)
{
    while (*dest++ = *src++) {
        ;
    }
    *dest = NUL;
}
```

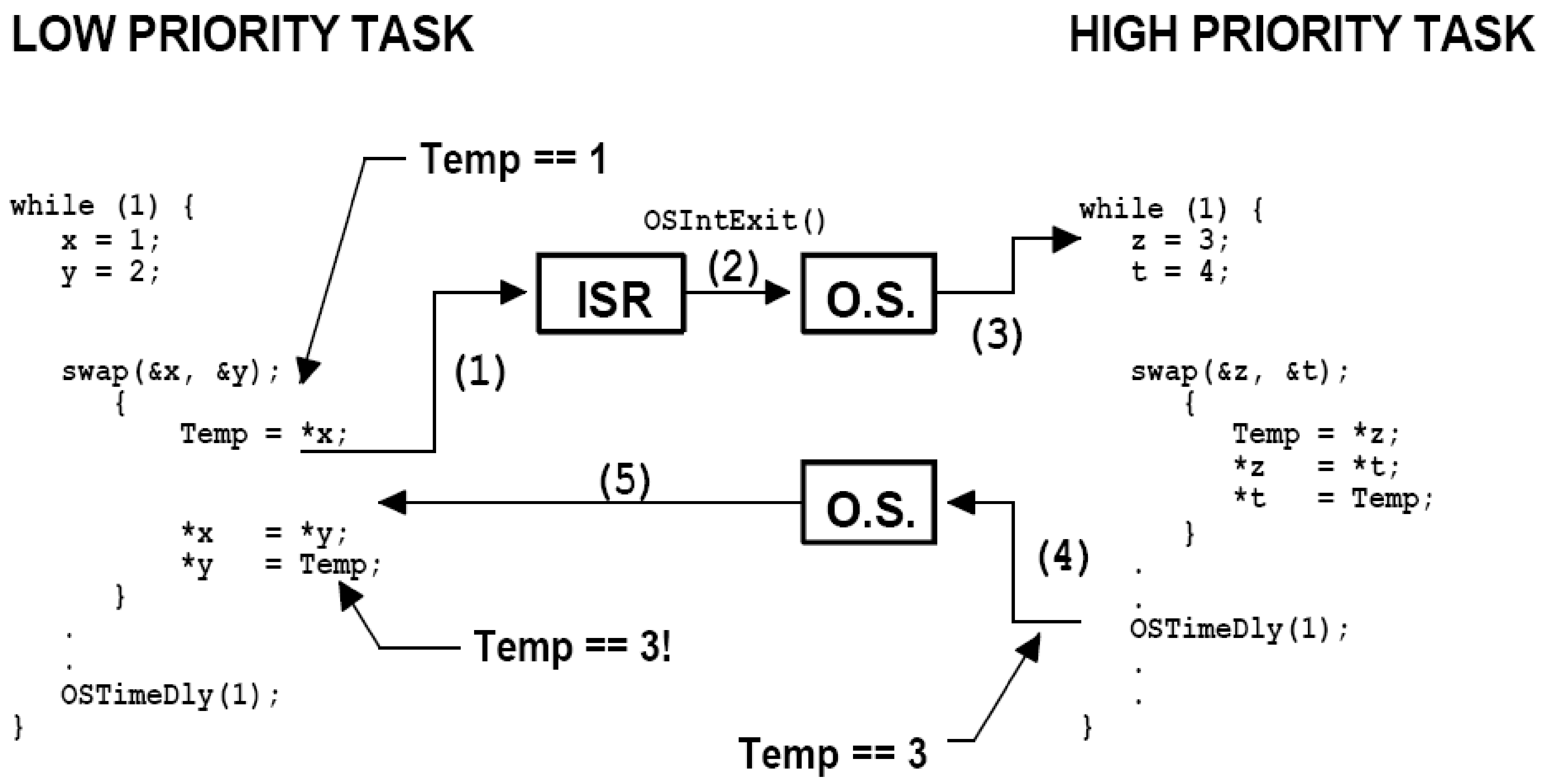
Reentrant Function



```
int Temp;

void swap(int *x, int *y)
{
    Temp = *x;
    *x   = *y;
    *y   = Temp;
}
```

Not-reentrant Function

Problem of reentrancy



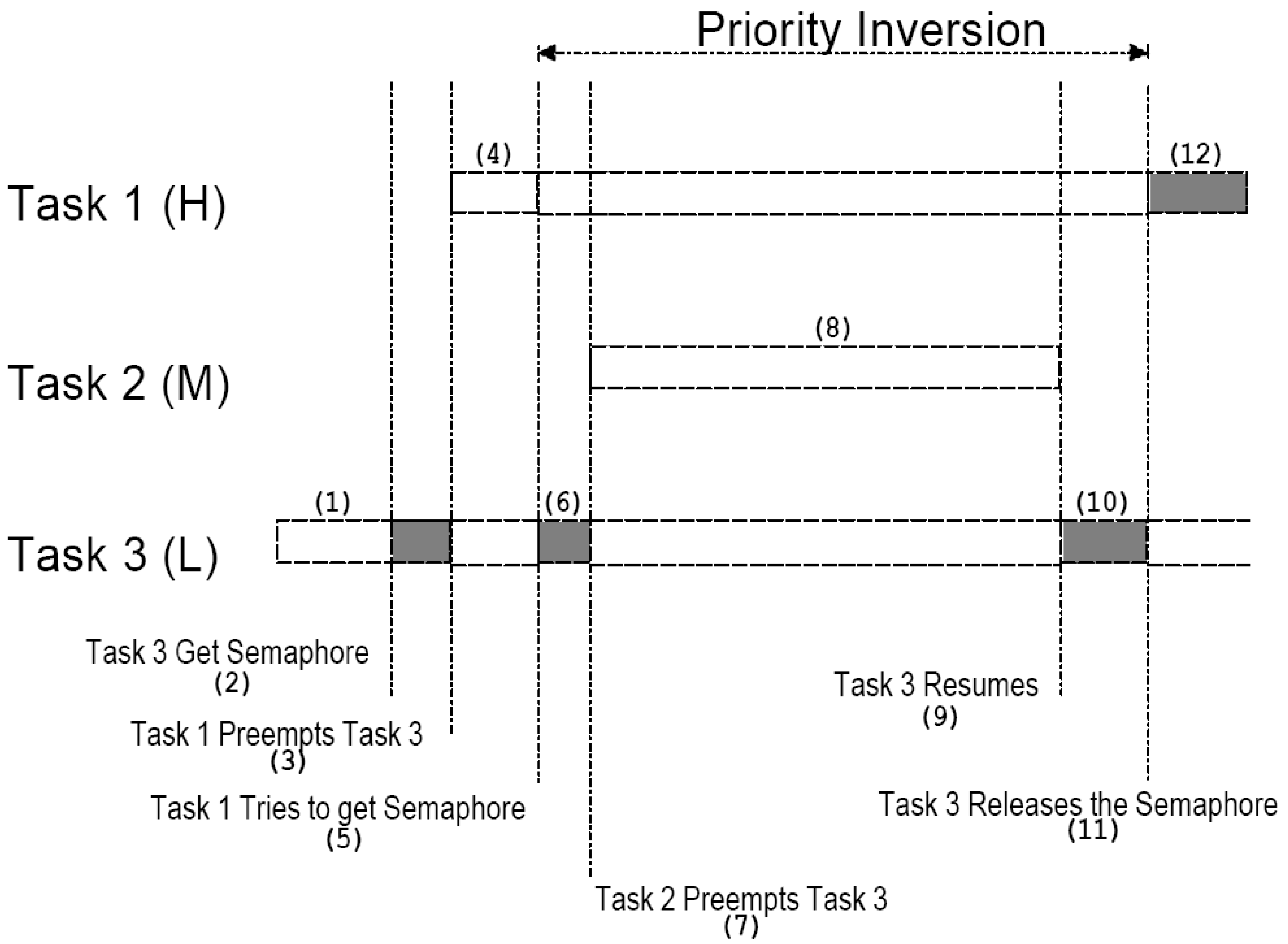
```
while (1) {
    x = 1;
    y = 2;

    swap(&x, &y);
      {
          Temp = *x;

          *x   = *y;
          *y   = Temp;
      }
    .
    .
    OSTimeDly(1);
}
```

```
while (1) {
    z = 3;
    t = 4;

    swap(&z, &t);
      {
          Temp = *z;
          *z   = *t;
          *t   = Temp;
      }
    .
    .
    OSTimeDly(1);
    .
    .
}
```

Priorities

- ***Task Priority*** - The more important the task, the higher the priority given to it.
- ***Static Priorities*** - priority of each task does not change during the application's execution. Each task is thus given a fixed priority at compile time.
- ***Dynamic Priorities*** - priority of tasks can be changed during the application's execution; each task can change its priority at run-time – **priority inversion problem**

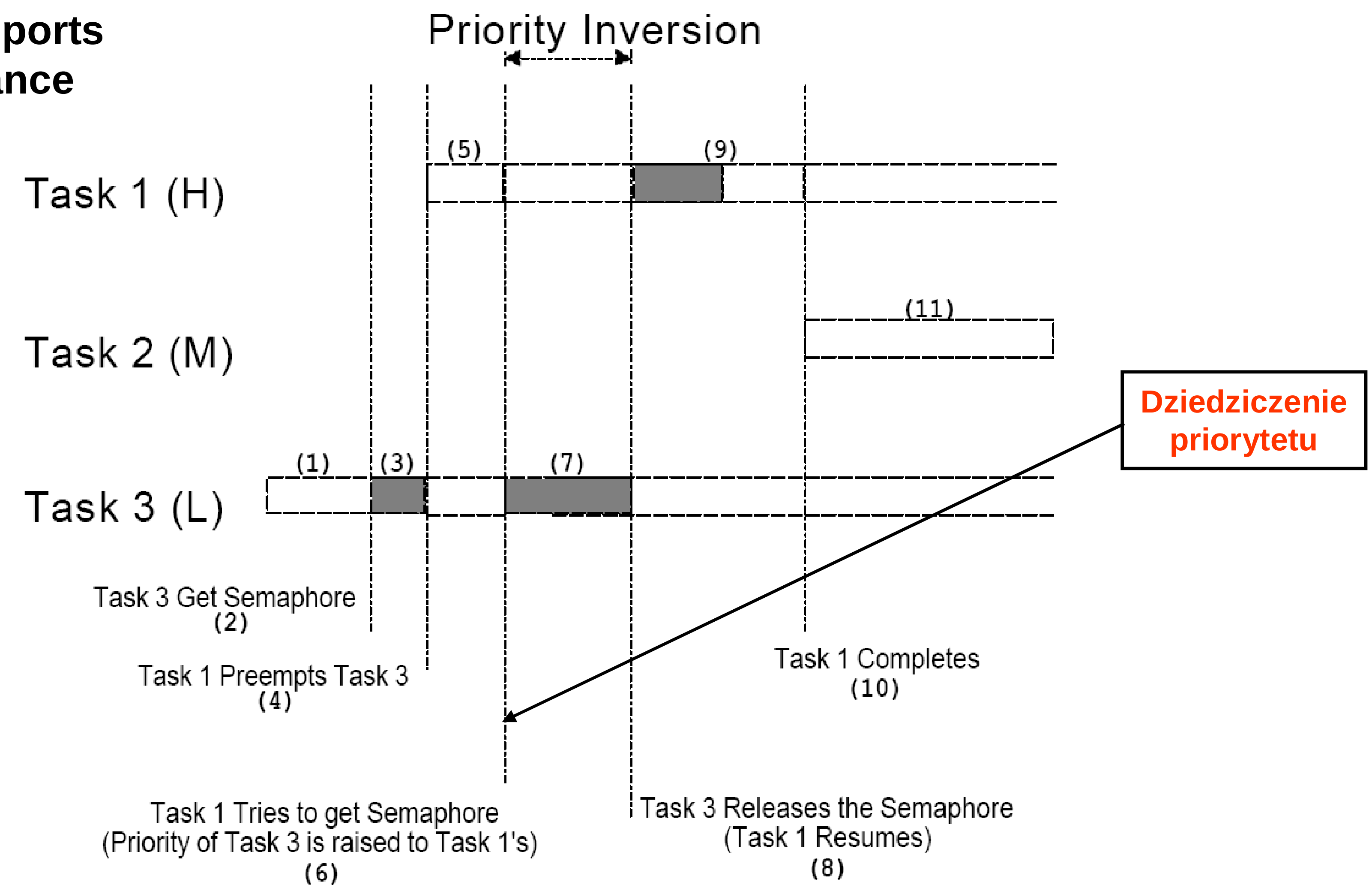
Priority inversion problem



Priority Inversion
Task 1 (H)
Task 2 (M)
Task 3 (L)
(1)
(4)
(6)
(8)
(10)
(12)
Task 3 Get Semaphore (2)
Task 1 Preempts Task 3 (3)
Task 1 Tries to get Semaphore (5)
Task 2 Preempts Task 3 (7)
Task 3 Resumes (9)
Task 3 Releases the Semaphore (11)

Priority inheritance

Kernel that supports priority inheritance



Priority Inversion
Task 1 (H)
Task 2 (M)
Task 3 (L)
(5)
(9)
(11)
(1)
(3)
(7)
Dziedziczenie priorytetu
Task 3 Get Semaphore (2)
Task 1 Preempts Task 3 (4)
Task 1 Completes (10)
Task 1 Tries to get Semaphore (Priority of Task 3 is raised to Task 1's) (6)
Task 3 Releases the Semaphore (Task 1 Resumes) (8)

Mutual Exclusion

The easiest way for tasks to communicate with each other is through shared data structures (easy when all tasks exist in a single address space):

- Global variables,
- Pointers,
- Buffers,
- Linked lists,
- Ring buffers, etc.

- While sharing data simplifies the exchange of information, you must ensure that each task has exclusive access to the data to avoid data corruption – this is called ***mutual exclusion.***

- Most common methods to obtain exclusive access to shared resources are:
  - Disabling Interrupts,
  - Test-And-Set,
  - Disabling Scheduling,
  - Using Semaphores.

Mutual Exclusion

- Disabling Interrupts

```
Disable interrupts;
Access the resource (read/write from/to variables);
Reenable interrupts;
```

- Test-And-Set

```
Disable interrupts;
if ('Access Variable' is 0) {
    Set variable to 1;
    Reenable interrupts;
    Access the resource;
    Disable interrupts;
    Set the 'Access Variable' back to 0;
    Reenable interrupts;
} else {
    Reenable interrupts;
    /* You don't have access to the resource, try back later; */
}
```

- Disabling Scheduling

```
void Function (void)
{
    OSSchedLock();
    .
    .     /* You can access shared data in here (interrupts are recognized) */
    .
    OSSchedUnlock();
}
```

Mutual Exclusion

- Semaphores

```
OS_EVENT *SharedDataSem;


void Function (void)
{
    INT8U err;


    OSSemPend(SharedDataSem, 0, &err);
    .
    .      /* You can access shared data in here (interrupts are recognized) */
    .
    OSSemPost(SharedDataSem);
}
```

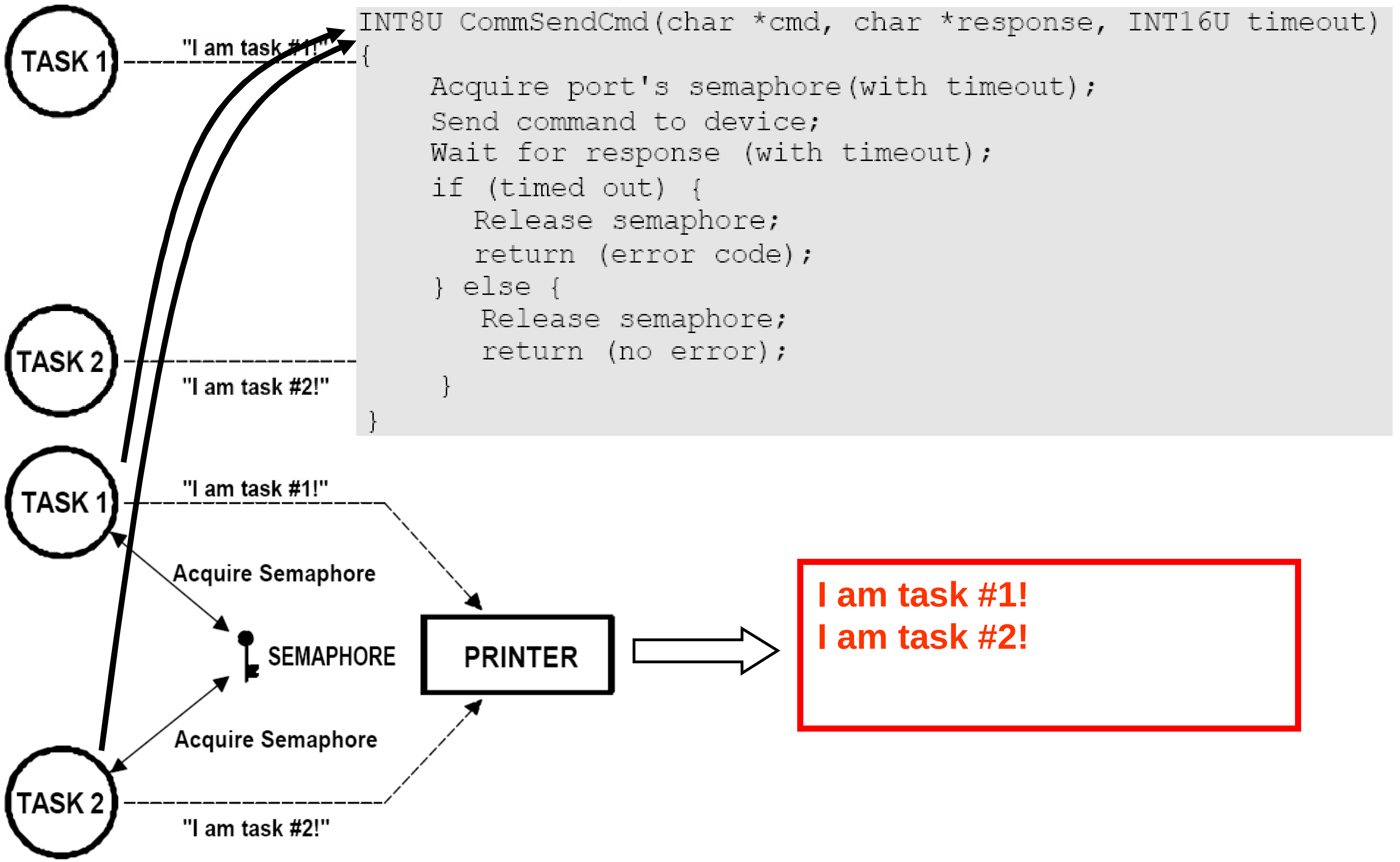
Semaphores

- ***Semaphore*** - protocol mechanism offered by most multitasking kernels. Semaphores are used to:
  - control access to a shared resource (mutual exclusion);
  - signal the occurrence of an event;
  - allow two tasks to synchronize their activities.
- A semaphore is a key that your code acquires in order to continue execution. If the semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner.
- Two types of semaphores – binary and counting (8-, 16- or 32-bit).
- Three operations that can be performed on semaphore:
  - INIT (CREATE)
  - WAIT (PEND)
  - SIGNAL (POST)

Semaphores

- The initial value of the semaphore must be provided when the semaphore is initialized. The waiting list of tasks is always initially empty.
- A task desiring the semaphore will perform a WAIT operation. If the semaphore is available (the semaphore value is greater than **0**), the semaphore value is decremented and the task continues execution. If the semaphore's value is **0**, the task performing a WAIT on the semaphore is placed in a waiting list. Most kernels allow you to specify a timeout; if the semaphore is not available within a certain amount of time, the requesting task is made ready to run and an error code (indicating that a timeout has occurred) is returned to the caller.
- A task releases a semaphore by performing a SIGNAL operation. If no task is waiting for the semaphore, the semaphore value is simply incremented. If any task is waiting for the semaphore, however, one of the tasks is made ready to run and the semaphore value is not incremented; the key is given to one of the tasks waiting for it. Depending on the kernel, the task which will receive the semaphore is either:
  - the highest priority task waiting for the semaphore, or
  - the first task that requested the semaphore (First In First Out, or FIFO).

Binary semaphores

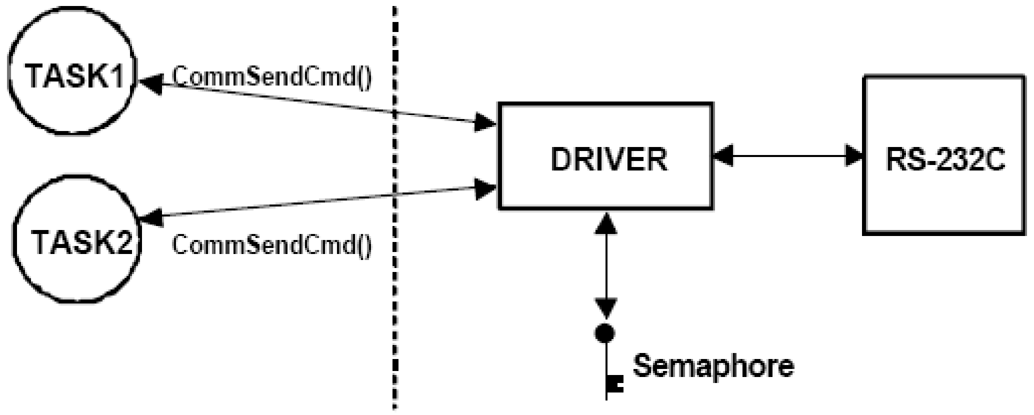


```
INT8U CommSendCmd(char *cmd, char *response, INT16U timeout)
{
    Acquire port's semaphore(with timeout);
    Send command to device;
    Wait for response (with timeout);
    if (timed out) {
      Release semaphore;
      return (error code);
    } else {
      Release semaphore;
      return (no error);
    }
}
```

I am task #1!
I am task #2!

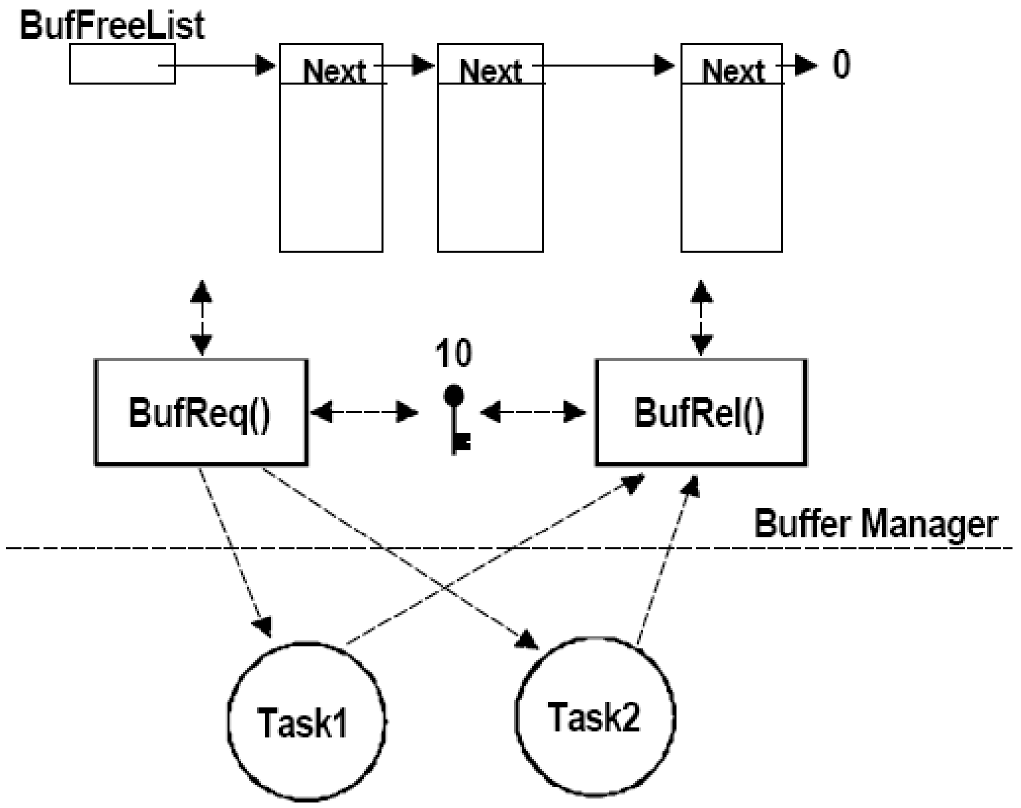
Binary semaphores

- Ukrywanie semaforów przed zadaniami



Counting Semaphores

- Buffer management using a counting semaphore
  - Buffer pool initially contains 10 buffers,
  - A task would obtain a buffer from the buffer manager by calling **BufReq()**.
  - When the buffer is no longer needed, the task would return the buffer to the buffer manager by calling **BufRel()**.



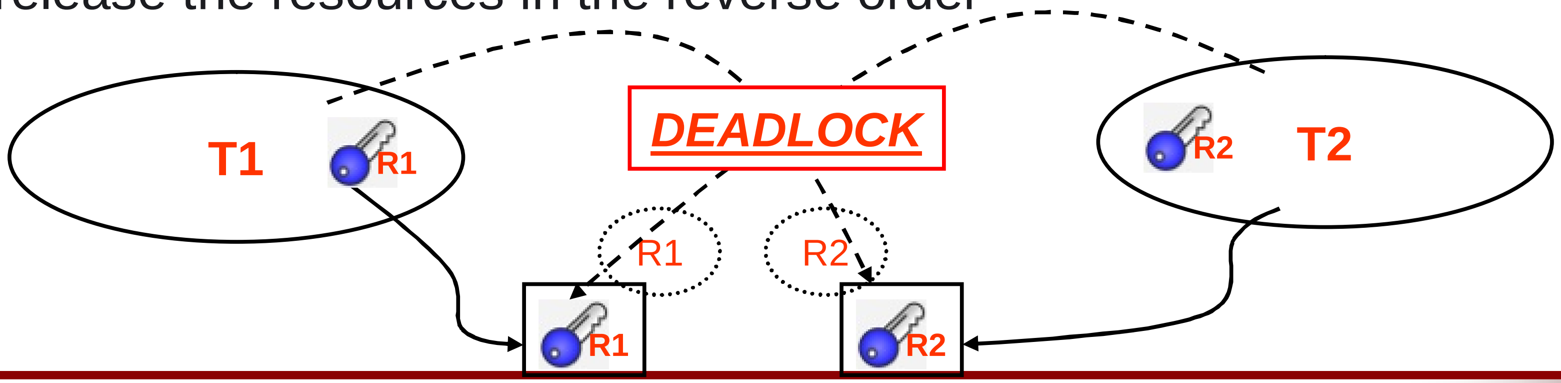
```
BUF *BufReq(void)
{
   BUF *ptr;

   Acquire a semaphore;
   Disable interrupts;
   ptr          = BufFreeList;
   BufFreeList = ptr->BufNext;
   Enable interrupts;
   return (ptr);
}


void BufRel(BUF *ptr)
{
   Disable interrupts;
   ptr->BufNext = BufFreeList;
   BufFreeList  = ptr;
   Enable interrupts;
   Release semaphore;
}
```

Semaphores

- Semaphores are often overused. The use of a semaphore to access a simple shared variable is overkill in most situations.
- The overhead involved in acquiring and releasing the semaphore can consume valuable time. You can do the job just as efficiently by disabling and enabling interrupts.
- Let's suppose that two tasks are sharing a 32-bit integer variable. The first task increments the variable while the other task clears it. If you consider how long a processor takes to perform either operation, you will realize that you do not need a semaphore to gain exclusive access to the variable. Each task simply needs to disable interrupts before performing its operation on the variable and enable interrupts when the operation is complete.
- A semaphore should be used, however, if the variable is a floating-point variable and the microprocessor doesn't support floating-point in hardware. In this case, the processing time involved in processing the floating-point variable could affect interrupt latency if you had disabled interrupts.

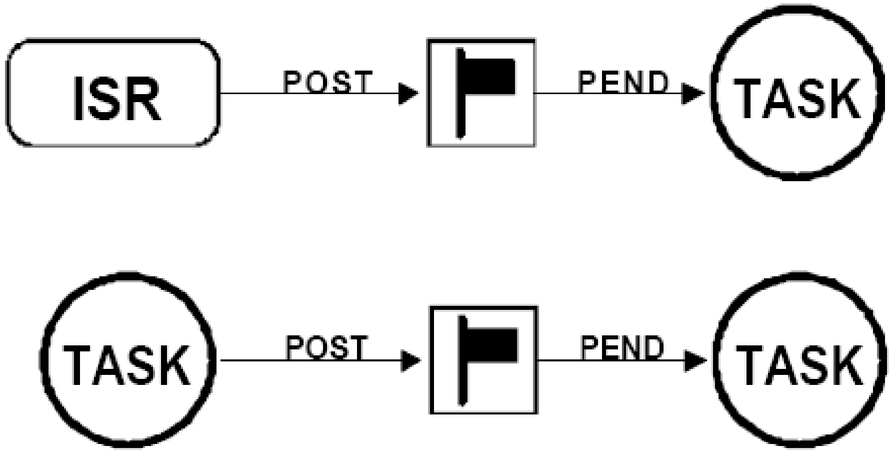
Deadlock Embrace

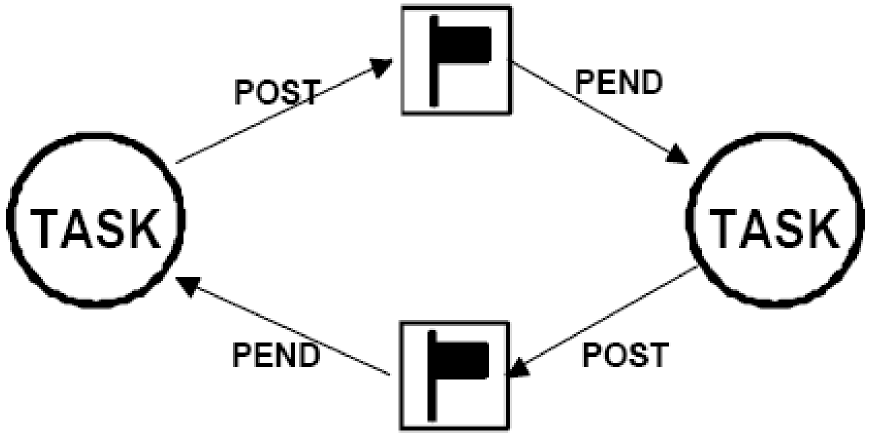
- ***Deadlock*** (***Deadky Embrace***) a situation in which two tasks are each unknowingly waiting for resources held by each other. If task T1 has exclusive access to resource R1 and task T2 has exclusive access to resource R2, then if T1 needs exclusive access to R2 and T2 needs exclusive access to R1, neither task can continue. They are deadlocked.
- The simplest way to avoid a deadlock is for tasks to:
  - acquire all resources before proceeding,
  - acquire the resources in the same order, and
  - release the resources in the reverse order



Unilateral / Bilateral Rendezvous

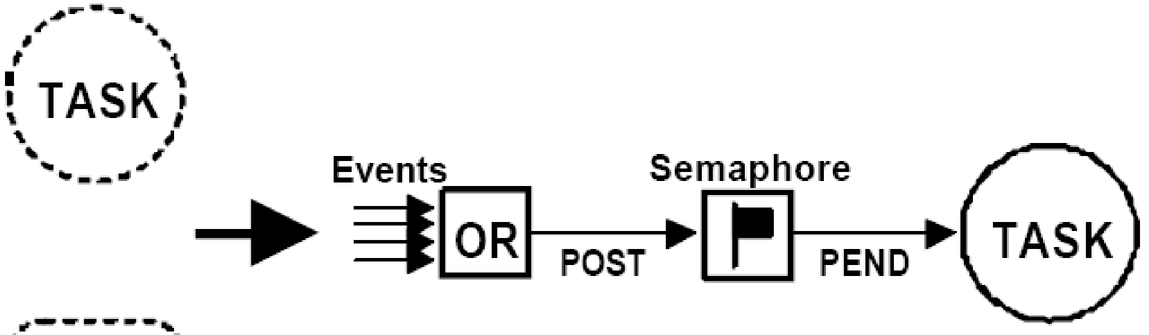
- Synchronizing tasks and ISRs - A task can be synchronized with an ISR, or another task when no data is being exchanged, by using a semaphore.
- Note that, in this case, the semaphore is drawn as a flag, to indicate that it is used to signal the occurrence of an event (rather than to ensure mutual exclusion, in which case it would be drawn as a key).
- When used as a synchronization mechanism, the semaphore is initialized to **0**. Using a semaphore for this type of synchronization is using what is called a ***unilateral rendezvous***. A task initiates an I/O operation and then waits for the semaphore. When the I/O operation is complete, an ISR (or another task) signals the semaphore and the task is resumed.
- Two tasks can synchronize their activities by using two semaphores. This is called a ***bilateral rendezvous***.



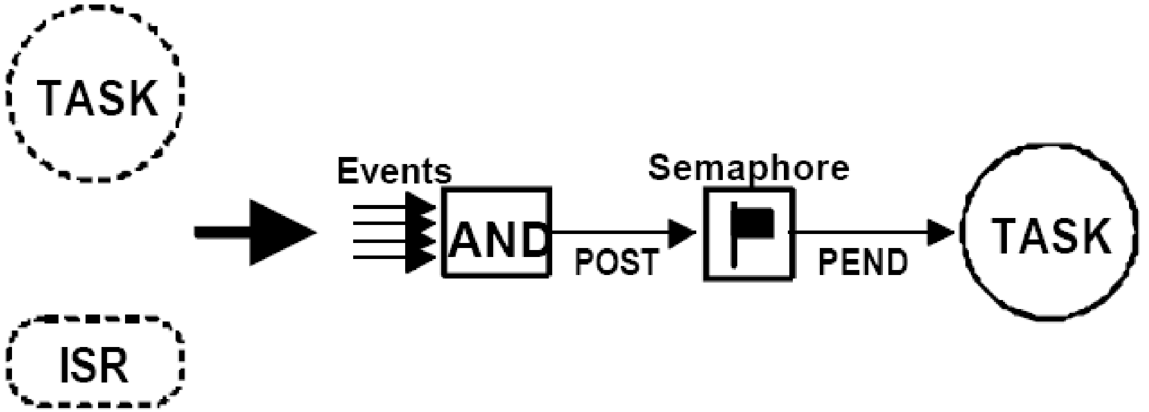


Conjunctive / Disjunctive Synchronization

- Event flags are used when a task needs to synchronize with the occurrence of multiple events. The task can be synchronized when
  - any of the events have occurred – *disjunctive synchronization* (logical OR),
  - all events have occurred - *conjunctive synchronization* (logical AND).



DISJUNCTIVE SYNCHRONIZATION



CONJUNCTIVE SYNCHRONIZATION

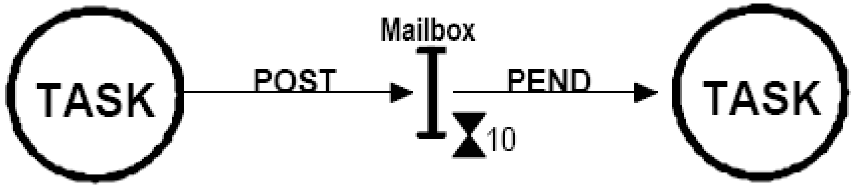
Intertask Communication

- It is sometimes necessary for a task or an ISR to communicate information to another task. This information transfer is called ***intertask communication***. Information may be communicated between tasks in two ways:
  - through global data;
  - by sending messages.

- When using global variables, each task or ISR must ensure that it has exclusive access to the variables.
  - If an ISR is involved, the only way to ensure exclusive access to the common variables is to disable interrupts.
  - If two tasks are sharing data each can gain exclusive access to the variables by using either disabling/enabling interrupts or through a semaphore (as we have seen). Note that a task can only communicate information to an ISR by using global variables.
- A task is not aware when a global variable is changed by an ISR unless the ISR signals the task by using a semaphore or by having the task regularly poll the contents of the variable. To correct this situation, you should consider using either a *message mailbox* or a *message queue*

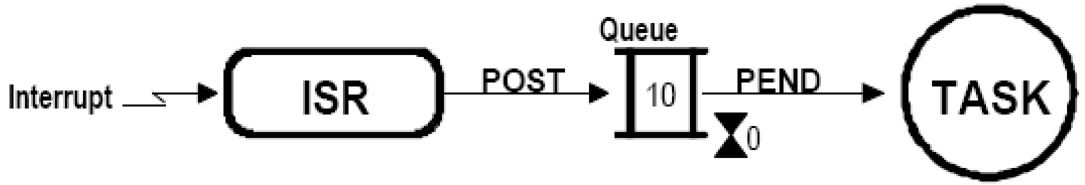
Message Mailbox

- ***Message Mailbox*** is typically a pointer size variable. Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into this mailbox. Similarly, one or more tasks can receive messages through a service provided by the kernel. <u>Both the sending task and receiving task will agree as to what the pointer is actually pointing to</u>.
- A waiting list is associated with each mailbox in case more than one task desires to receive messages through the mailbox. A task desiring to receive a message from an empty mailbox will be suspended and placed on the waiting list until a message is received.
- Typically, the kernel will allow the task waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting task is made ready-to-run and an error code (indicating that a timeout has occurred) is returned to it.
- When a message is deposited into the mailbox, either the highest priority task waiting for the message is given the message (called *priority-based*) or the first task to request a message is given the message (called *First-In-First-Out*, or FIFO).



Message Queue

- Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into a ***message queue***. Similarly, one or more tasks can receive messages through a service provided by the kernel. Both the sending task and receiving task will agree as to what the pointer is actually pointing to. Generally, the first message inserted in the queue will be the first message extracted from the queue (FIFO).
- As with the mailbox, a waiting list is associated with each message queue in case more than one task is to receive messages through the queue. A task desiring to receive a message from an empty queue will be suspended and placed on the waiting list until a message is received.
- Typically, the kernel will allow the task waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting task is made ready-to-run and an error code (indicating a timeout occurred) is returned to it.
- When a message is deposited into the queue, either the highest priority task or the first task to wait for the message will be given the message.

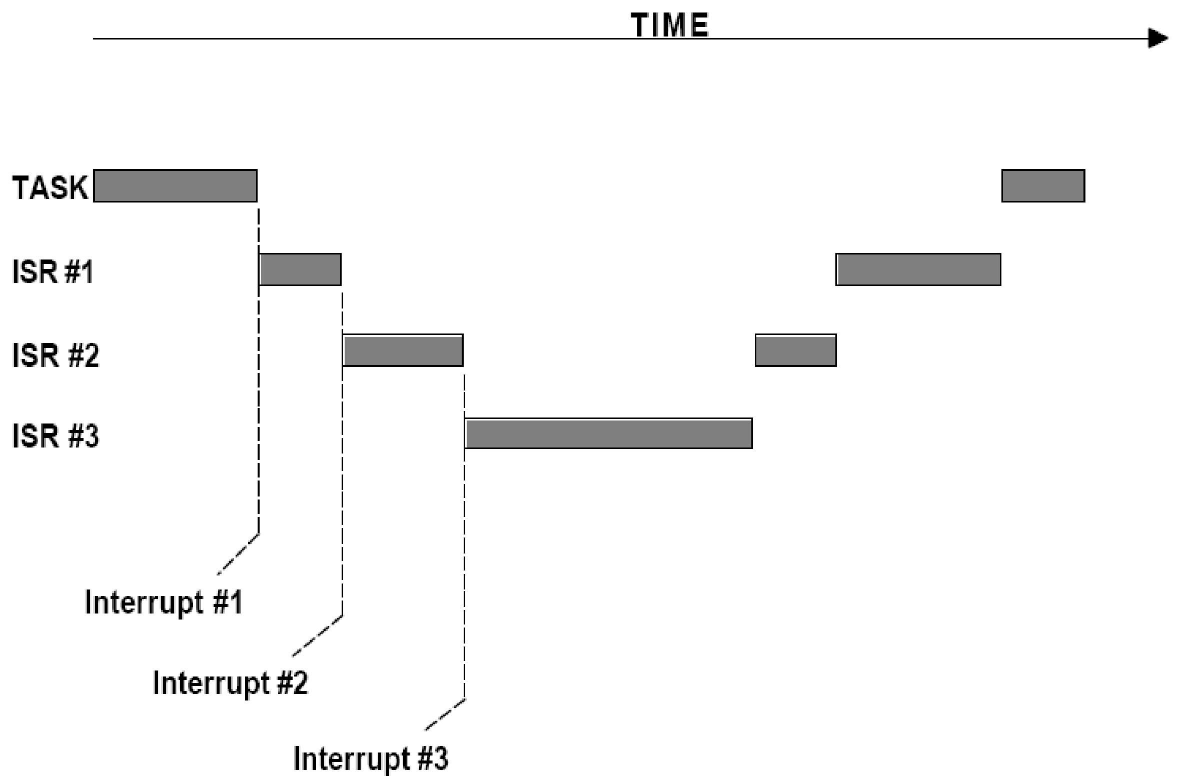


Interrupt Service Routines

- An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred. When an interrupt is recognized, the CPU saves part (or all) of its context (i.e. registers) and jumps to a special subroutine called an *Interrupt Service Routine*.
- The ISR processes the event and upon completion of the ISR, the program returns to:
  - The background for a foreground/background system.
  - The interrupted task for a non-preemptive kernel.
  - The highest priority task ready-to-run for a preemptive kernel.

- Interrupts allow a microprocessor to process events when they occur. This prevents the microprocessor from continuously *polling* an event to see if this event has occurred. Microprocessors allow interrupts to be ignored and recognized through the use of two special instructions: *disable interrupts* and *enable interrupts*, respectively.
- In a real-time environment, interrupts should be disabled as little as possible. Disabling interrupts affects interrupt latency and also, disabling interrupts may cause interrupts to be missed.

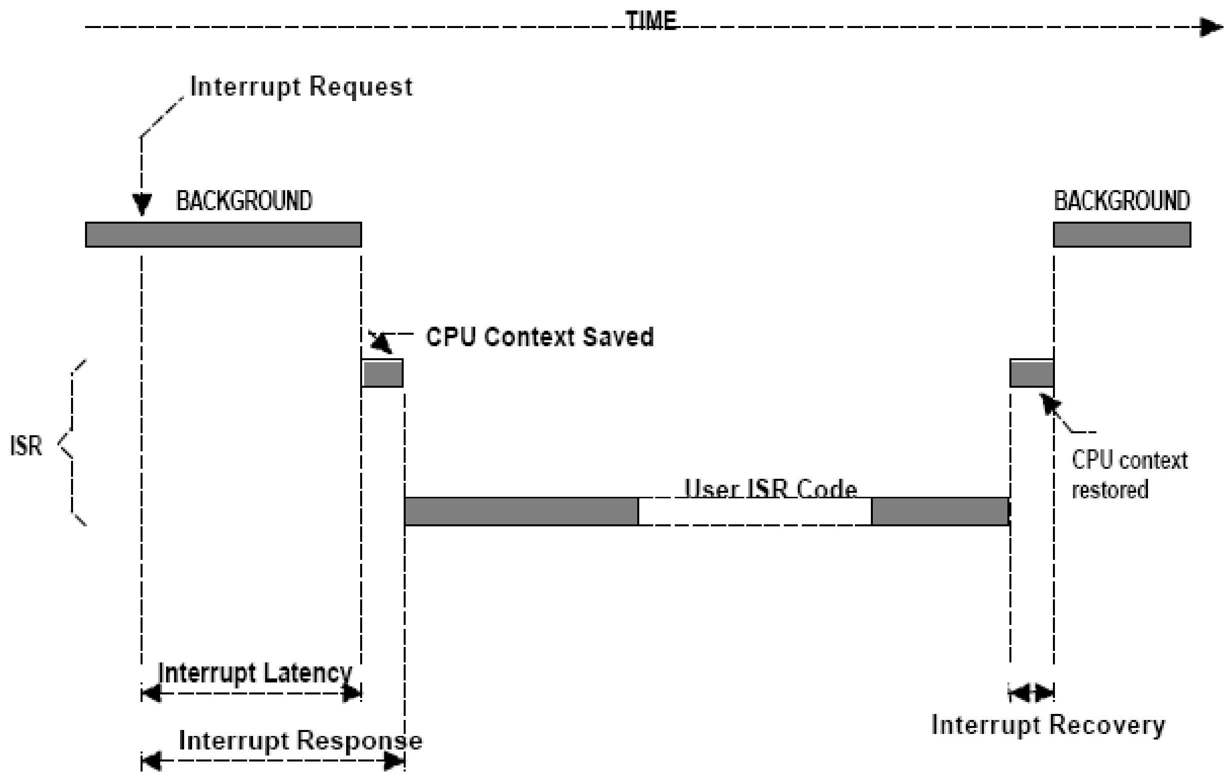
Interrupt Service Routines



- Processors generally allow interrupts to be *nested*. This means that while servicing an interrupt, the processor will recognize and service other (more important) interrupts.

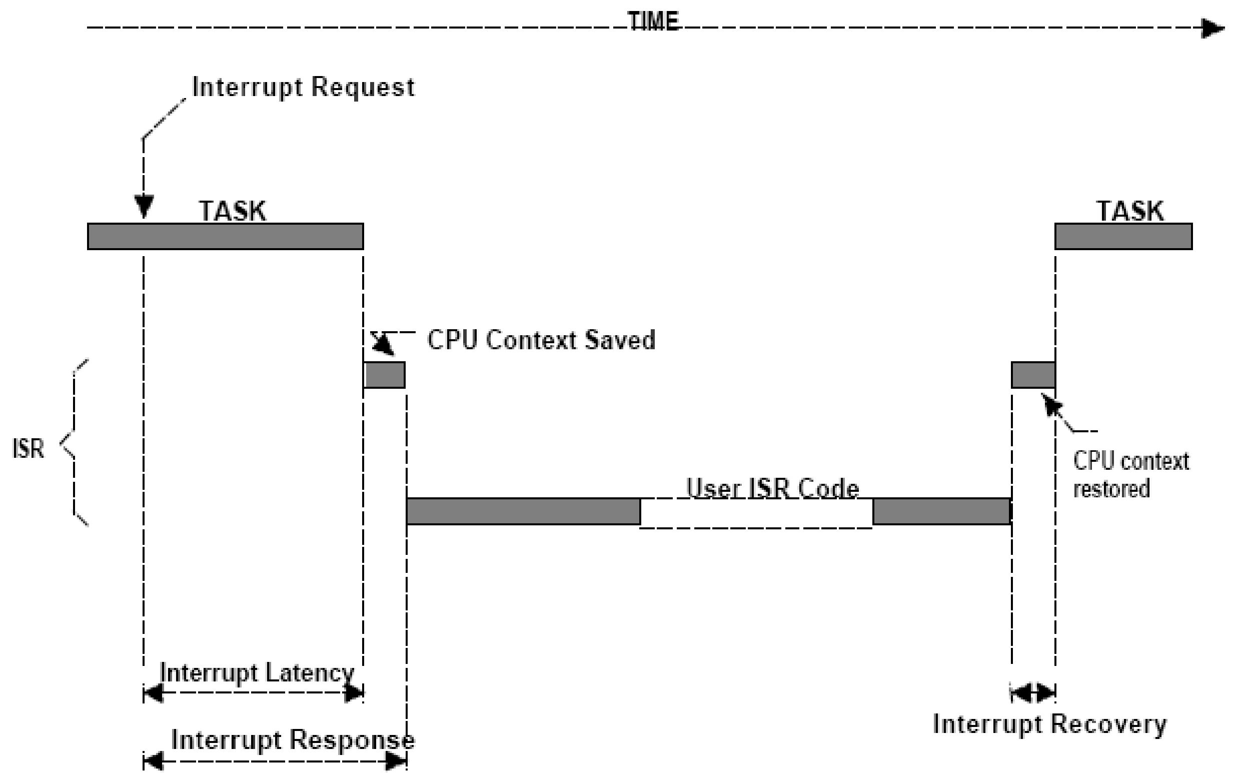
Interrupt Latency, Response, and Recovery

Foreground/Background System



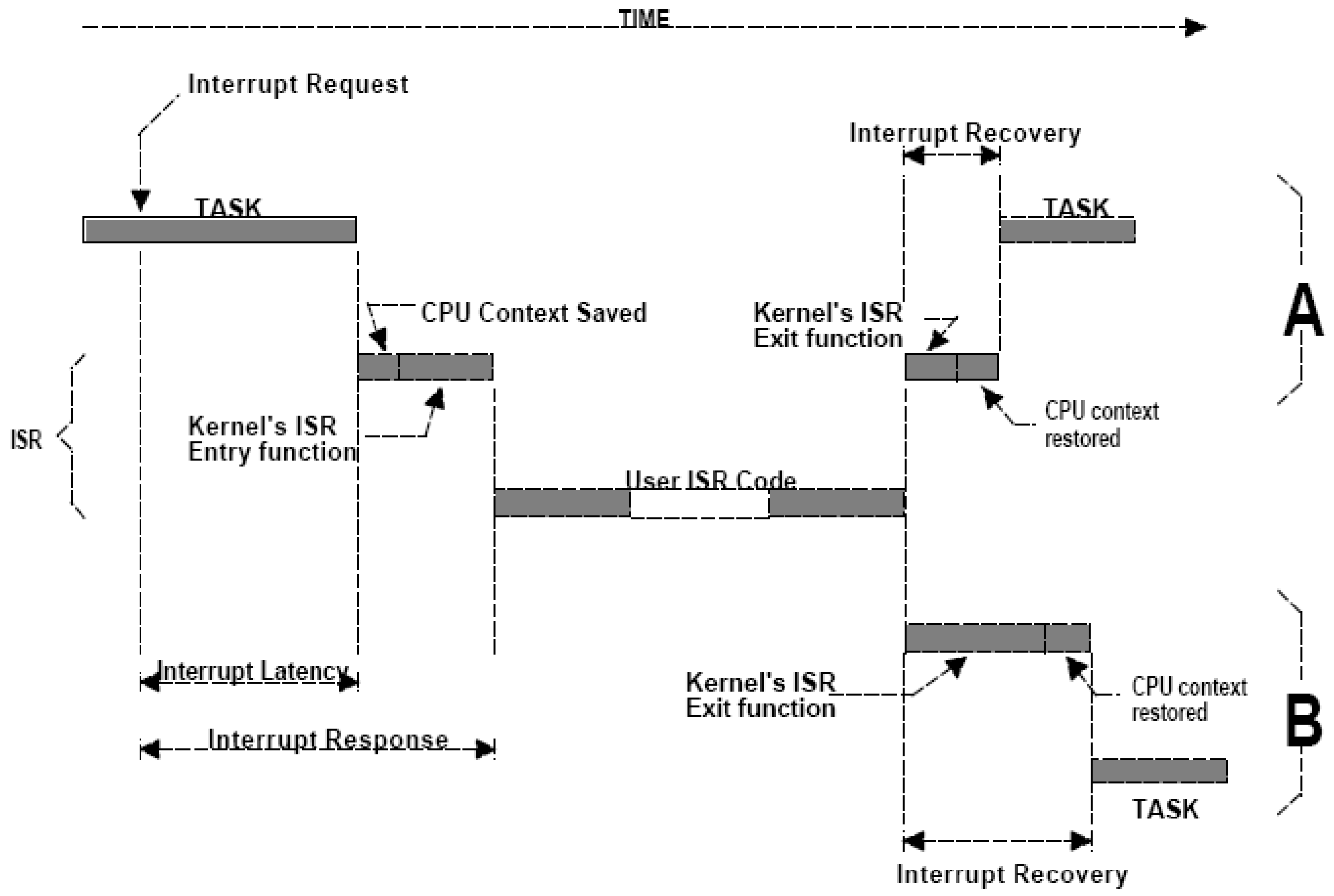
Interrupt Latency, Response, and Recovery

Non-preemptive Kernel



Interrupt Latency, Response, and Recovery

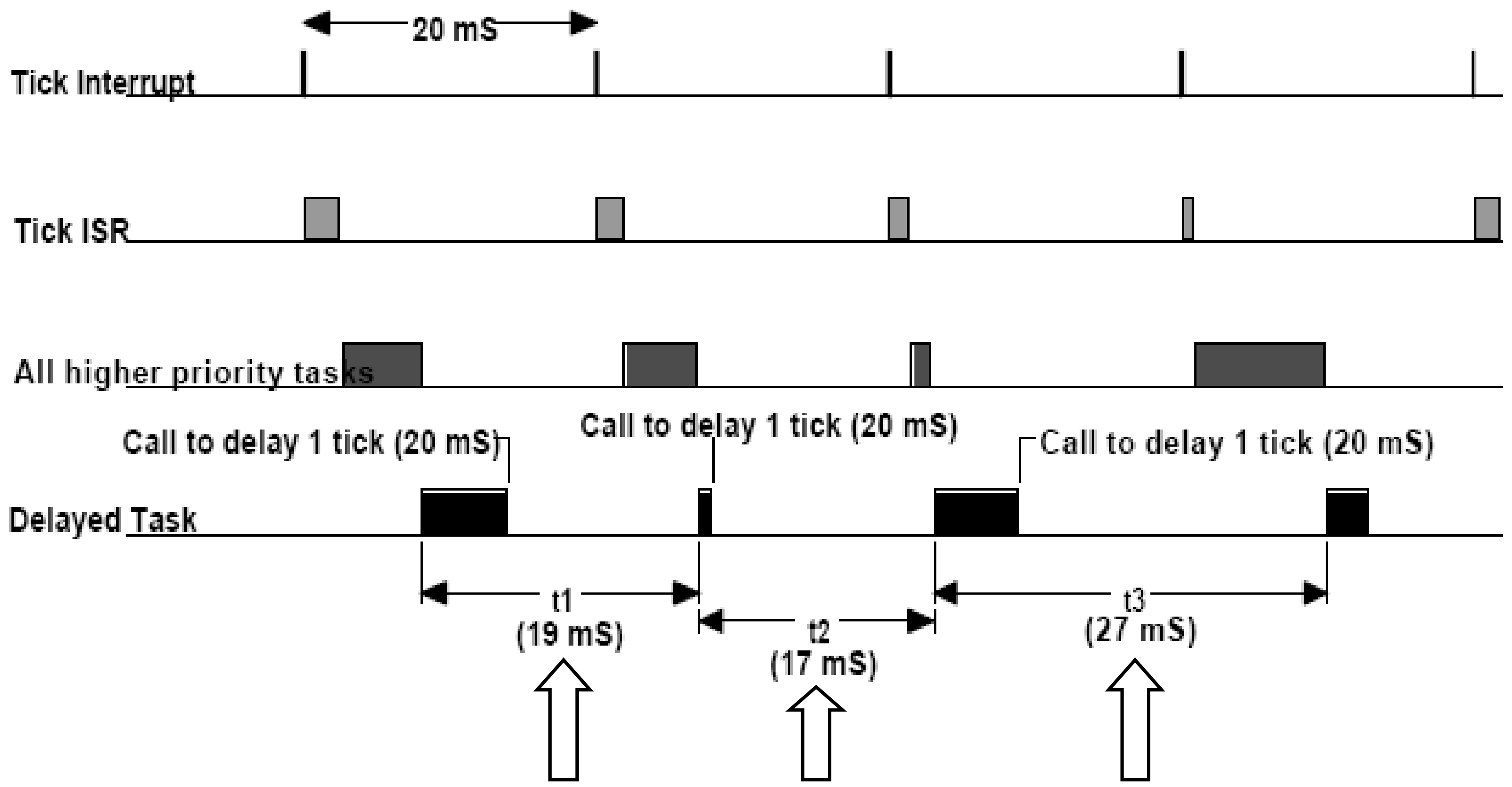
Preemptive Kernel



RTOS Clock Tick

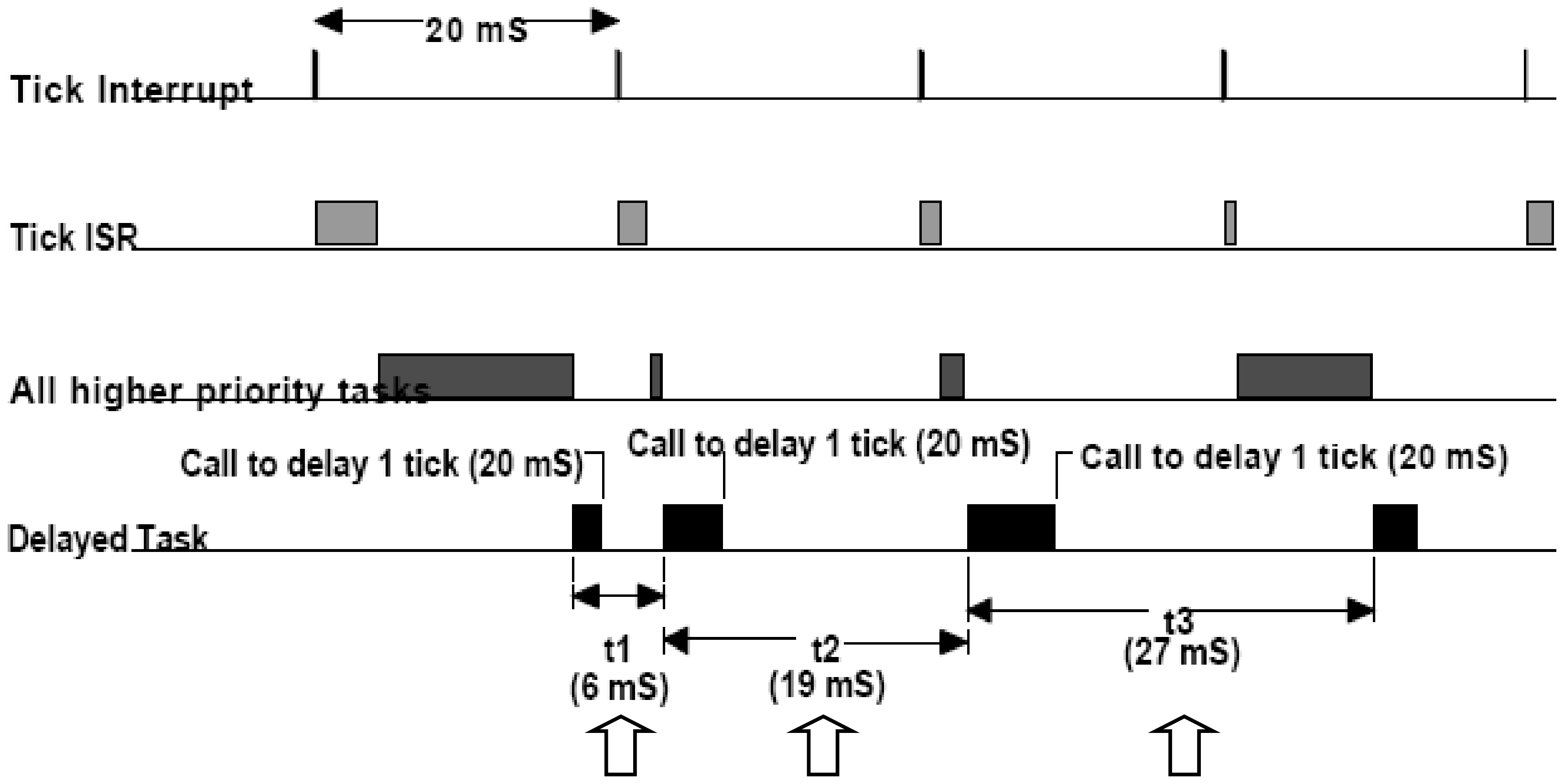
- ***Clock Tick*** – special interrupt that occurs periodically. This interrupt can be viewed as the system's heartbeat. The time between interrupts is application specific and is generally between 10 and 200 mS. The clock tick interrupt allows a kernel to delay tasks for an integral number of clock ticks and to provide timeouts when tasks are waiting for events to occur. The faster the tick rate, the higher the overhead imposed on the system.
- All kernels allow tasks to be delayed for a certain number of clock ticks. The resolution of delayed tasks is 1 clock tick, however, this does not mean that its accuracy is 1 clock tick.

RTOS Clock Tick


20 mS
Tick Interrupt
Tick ISR
All higher priority tasks
Call to delay 1 tick (20 mS)
Call to delay 1 tick (20 mS)
Call to delay 1 tick (20 mS)
Delayed Task
t1
(19 mS)
t2
(17 mS)
t3
(27 mS)

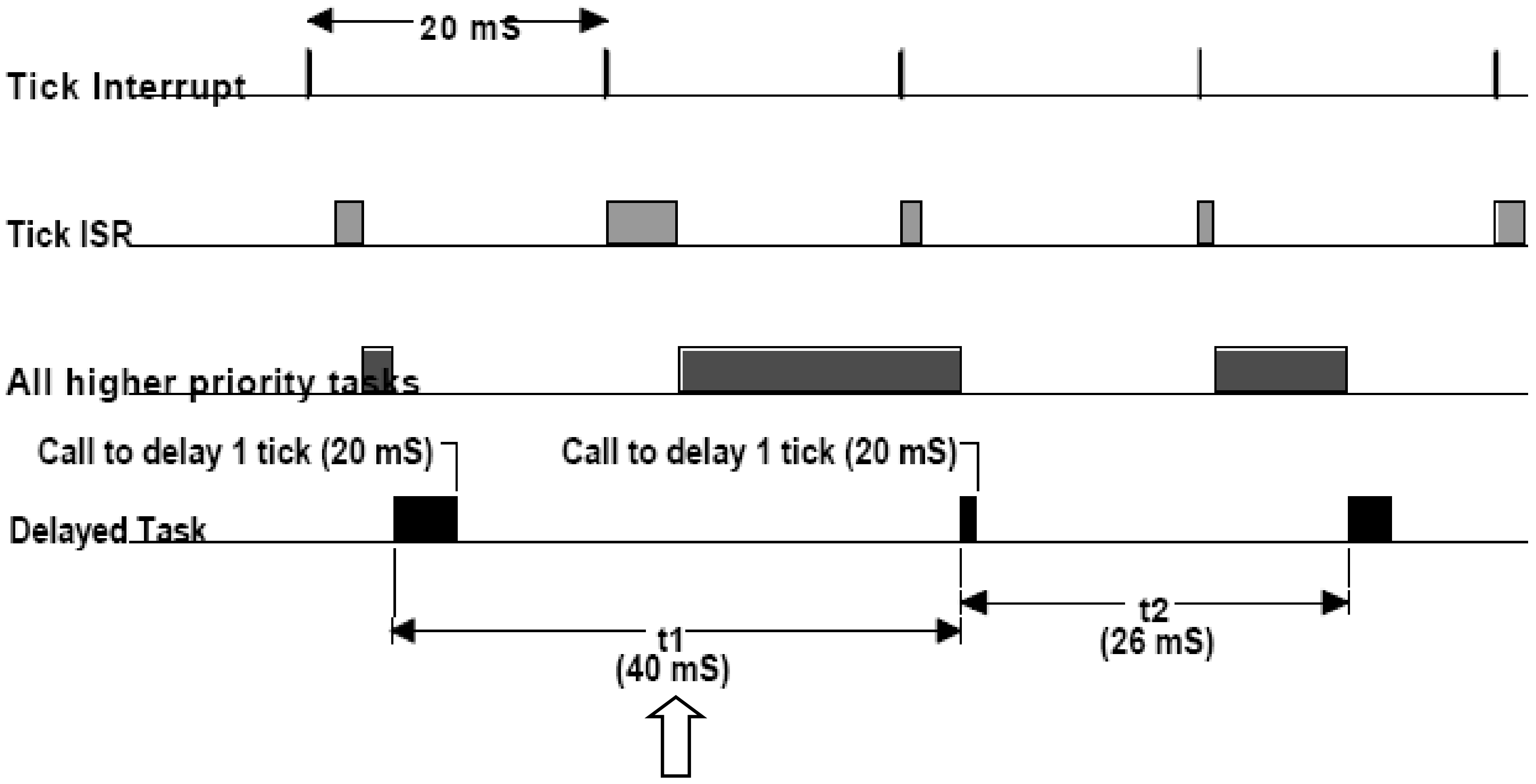
The task attempts to delay for 20 mS but because of its priority, actually executes at varying intervals. This will thus cause the execution of the task to *jitter.*

RTOS Clock Tick


20 mS
Tick Interrupt
Tick ISR
All higher priority tasks
Call to delay 1 tick (20 mS)
Call to delay 1 tick (20 mS)
Call to delay 1 tick (20 mS)
Delayed Task
t1
(6 mS)
t2
(19 mS)
t3
(27 mS)

The execution times of all higher-priority tasks and ISRs are slightly less than one tick. If the task delays itself just before a clock tick, the task will execute again almost immediately! Because of this, if you need to delay a task for at least 1 clock tick, you must specify one extra tick. In other words, if you need to delay a task for at least 5 ticks, you must specify 6 ticks!

RTOS Clock Tick

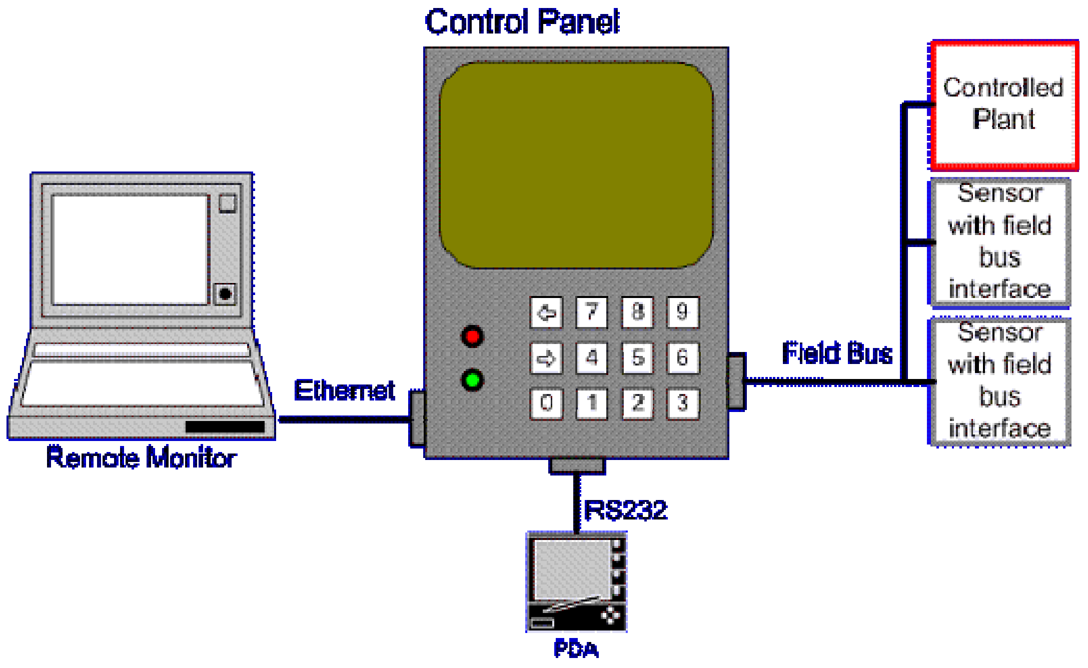

20 mS
Tick Interrupt
Tick ISR
All higher priority tasks
Call to delay 1 tick (20 mS)
Call to delay 1 tick (20 mS)
Delayed Task
t1
(40 mS)
t2
(26 mS)

The execution times of all higher-priority tasks and ISRs extend beyond one clock tick. In this case, the task that tries to delay for 1 tick will actually execute 2 ticks later! In this case, the task missed its deadline. This might be acceptable in some applications, but in most cases it isn't.

Comparision

	Foreground/Background	Non-Preemptive Kernel	Preemptive Kernel
Interrupt Latency (Time)	MAX (Longest instruction, User interrupt disable) + Vector to ISR	MAX(Longest instruction, User interrupt disable, Kernel interrupt disable) + Vector to ISR	MAX(Longest instruction, User interrupt disable, Kernel interrupt disable) + Vector to ISR
Interrupt Response (Time)	Interrupt latency + Save CPU's context	Interrupt latency + Save CPU's context	Interrupt latency + Save CPU's context + Kernel ISR entry function
Interrupr Recovery (Time)	Restore background's context + Return from interrupt	Restore task's context + Return from interrupt	Find highest priority task + Restore highest priority task's context + Return from interrupt
Task Response (Time)	Background	Longest task + Find highest priority task + Context switch	Find highest priority task + Context switch
ROM size	Application code	Application code + Kernel code	Application code + Kernel code
RAM size	Application code	Application code + Kernel RAM +SUM(Task stacks + MAX(ISR stack))	Application code + Kernel RAM +SUM(Task stacks + MAX(ISR stack))
Services available?	Application code must provide	Yes	Yes

Przykładowy system



- The system consists of:
  - An embedded computer within a control terminal.
  - Two fieldbus networked sensors.
  - The plant being controlled (could be anything, motor, heater, etc.). This is connected on the same fieldbus network.
  - A matrix keypad that is scanned using general purpose IO.
  - Two LED indicators.
  - An LCD display.
  - An embedded WEB server to which a remote monitoring computer can attach.
  - An RS232 interface to a configuration utility that runs on a PDA.

Stawiane wymagania / zadania

- **Plant Control**
  - Each control cycle shall perform the following sequence:
    - Transmit a frame on the fieldbus to request data from the networked sensors.
    - Wait to receive data from both sensors.
    - Execute the control algorithm.
    - Transmit a command to the plant.
  - The control function of the embedded computer shall transmit a request every 10ms exactly, and the resultant command shall be transmitted within 5ms of this request. The control algorithm is reliant on accurate timing, it is therefore paramount that these timing requirements are met.
- **Local Operator Interface [keypad and LCD]**
  - The keypad and LCD can be used by the operator to select, view and modify system data. The operator interface shall function while the plant is being controlled. To ensure no key presses are missed the keypad shall be scanned at least every 15ms. The LCD shall update within 50ms of a key being pressed.

Stawiane wymagania / zadania

- **LED**
  - The LED shall be used to indicate the system status. A flashing green LED shall indicate that the system is running as expected. A flashing red LED shall indicate a fault condition. The correct LED shall flash on and off once ever second. This flash rate shall be maintained to within 50ms.
- **RS232 PDA Interface**
  - The PDA RS232 interface shall be capable of viewing and accessing the same data as the local operator interface, and the same timing constraints apply - discounting any data transmission times.
- **TCP/IP Interface**
  - The embedded WEB server shall service HTTP requests within one second

Stawiane wymagania / zadania - podział

- The timing requirements of the hypothetical system can be split into three categories:
  - **Strict timing** - the plant control
    - The control function has a very strict timing requirement as it must execute every 10ms.
  - **Flexible timing** - the LED
    - While the LED outputs have both maximum and minimum time constraints, there is a large timing band within which they can function.
  - **Deadline only timing** - the human interfaces
    - This includes the keypad, LCD, RS232 and TCP/IP Ethernet communications.
    - The human interface functions have a different type of timing requirement as only a maximum limit is specified. For example, the keypad must be scanned at least every 10ms, but any rate up to 10ms is acceptable.

Rozwiązanie 1: SUPER LOOP

- Each component of the application is represented by a function that executes to completion.
- Ideally a hardware timer would be used to schedule the time critical plant control function. However, having to wait for the arrival of data and the complex calculation performed make the control function unsuitable for execution within an interrupt service routine.
- Prioritisation can be introduced with frequency and order in which components are called within the loop.

Rozwiązanie 1:
SUPER LOOP

- ☺ Small code size.
- ☺ No reliance on third party source code.
- ☺ No RTOS RAM, ROM or processing overhead.
- ☹ Difficult to cater for complex timing requirements.
- ☹ Does not scale well without a large increase in complexity.
- ☹ Timing hard to evaluate or maintain due to the interdependencies between the different functions.

SUPER LOOP - The Plant Control Function

- The control function can be represented by the following pseudo code:

```
void PlantControlCycle( void ) {
  TransmitRequest();
  WaitForFirstSensorResponse();

  if ( Got data from first sensor ) {
    WaitForSecondSensorResponse();

    if( Got data from second sensor ) {
      PerformControlAlgorithm();
      TransmitResults();
    }
  }
}
```

SUPER LOOP - The Human Interface Functions

- Keypad, LCD, RS232 communications and embedded WEB server.

```
int main( void ) {
  Initialise();
  for( ;; ) {
    ScanKeypad();
    UpdateLCD();
```

Two assumptions:

- Comunications IO is buffered by interrupt service routines so peripherals do not require polling,
- Individual function calls within the loop execute quickly enough for all the maximum timing requirements to be met.

SUPER LOOP - Scheduling the Plant Control Function

- Control function cannot be simply called from a 10ms timer interrupt – it is too long. We need some temporal control. For example:

```c
int TimerExpired;
// Configured to execute every 10ms.
void TimerInterrupt( void ) {
  TimerExpired = true;
}

int main( void ) {
  Initialise();

  for( ;; ) {
    if( TimerExpired ) {
      PlantControlCycle();
      TimerExpired = false;
      ScanKeypad();
      UpdateLCD();

      // The LEDs could use a count of
      // the number of interrupts, or a
      // different timer.
      ProcessLEDs();

      // Comms buffers must be large
      // enough to hold 10ms worth of
      // data.
      ProcessRS232Characters();

      ProcessHTTPRequests();
    }
    // The processor can be put to sleep
    // here provided it is woken by any
    // interrupt.
  }
  // Should never get here.
  return 0;
}
```

... but this is not an acceptable solution…

SUPER LOOP - Scheduling the Plant Control Function

- Relies on every function maximum / minimum execution time, not very maintainable:
  - A delay or fault on the field bus results in an increased execution time of the plant control function – problem with interface functions.
  - Executing all the functions each cycle could result in a breach of the control cycle timing.
- Jitter in the execution time may cause cycles to be missed (ex. the execution time of `ProcessHTTPRequests()` could be negligible when no HTTP requests have been received, but quite lengthy when a page was being served).
- The communication buffers are only serviced once per cycle necessitating their length to be larger than would otherwise be necessary.

SUPER LOOP - korekty

- Allowing each function to execute in its entirety takes too long => split each function into a number of states. Only one state is executed each call.

```
// Define the states
// for the control cycle function.
typdef enum eCONTROL_STATES {
  eStart, // Start new cycle.
  eWait1, // First sensor response.
  eWait2  // Second sensor response.
} eControlStates;

void PlantControlCycle( void ) {
  static eControlState eState = eStart;

  switch( eState ) {
    case eStart :
      TransmitRequest();
      eState = eWait1;
    break;
```

```
    case eWait1;
      if( Got data from first sensor ) {
        eState = eWait2;
      }
      // How are time outs to be handled?
    break;

    case eWait2;
      if( Got data from first sensor ) {
        PerformControlAlgorithm();
        TransmitResults();
        eState = eStart;
      }
      // How are time outs to be handled?
    break;
  }
}
```

SUPER LOOP - korekty

- This function is now structurally more complex, and introduces further scheduling problems.
- The code itself will become harder to understand as extra states are added - for example to handle timeout and error conditions.

SUPER LOOP - korekty

- The granularity of the timer - a shorter timer interval will give more flexibility.
- Implementing the control function as a state machine (an in so doing making each call shorter) may allow it to be called from a timer interrupt.
- timer interval will have to be short enough to ensure the function gets called at a frequency that meets its timing requirements.
- Alternatively
  – Infinite loop solution could be modified to call different functions on each loop - with the high priority control function called more frequently.

SUPER LOOP - korekty

```
int main( void ) {
  int Counter = -1;
  Initialise();

  // Function is implemented as a state
  // machine, execution is much quicker.
  // Timer frequency has been raised.
  for( ;; ) {
    if( TimerExpired ) {
      Counter++;
      switch( Counter ) {
        case 0 :
          ControlCycle();
          ScanKeypad();
        break;

        case 1 :
          UpdateLCD();
        break;

        case 2 :
          ControlCycle();
          ProcessRS232Characters();
        break;

        case 3 :
          ProcessHTTPRequests();
          // Go back to start
          Counter = -1;
        break;
      }
      TimerExpired = false;
    }
  }
  // Should never get here.
  return 0;
}
```

SUPER LOOP - korekty

- More intelligence can be introduced by means of event counters, whereby the lower priority functionality is only called if an event has occurred that requires servicing:

```
for( ;; ) {
  if( TimerExpired ) {
    Counter++;

    //Process the control cycle
    // every other loop.
    switch( Counter ) {
      case 0 :
        ControlCycle();
      break;

      case 1 :
        Counter = -1;
        break;
    }

    // Process just one of the other functions.
    // Only process a function if there is
    // something to do. EventStatus() checks
    // for events since the last iteration.
```

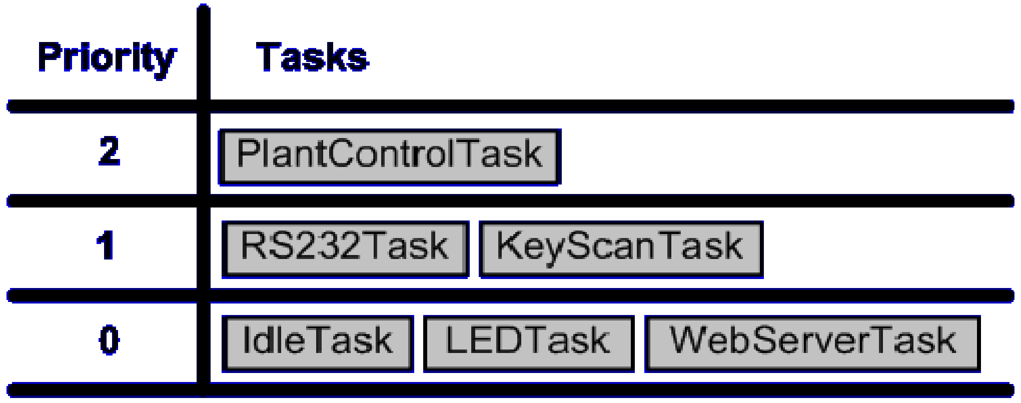
```
    switch( EventStatus() ) {
      case EVENT_KEY :
        ScanKeypad();
        UpdateLCD();
      break;

      case EVENT_232 :
        ProcessRS232Characters();
      break;

      case EVENT_TCP :
        ProcessHTTPRequests();
      break;
    }
    TimerExpired = false;
  }
}
```

SUPER LOOP - korekty

- Processing events in this manner will reduce wasted CPU cycles but the design will still exhibit jitter in the frequency at which the control cycle executes.

Rozwiązanie 2:
Traditional Preemptive Multitasking System

- A separate task is created for each part of the system (when it is able to exist in isolation, or is having a particular timing requirement).
- Tasks will block until an event indicates that processing is required. Events can either be external (ex. key being pressed), or internal (ex. timer expiring).
- Priorities - allocated to tasks in accordance to their timing requirements. The stricter the timing requirement the higher the priority



Priority	Tasks
2	PlantControlTask
1	RS232Task, KeyScanTask
0	IdleTask, LEDTask, WebServerTask

Rozwiązanie 2: Traditional Preemptive Multitasking System

- **Concept of Operation**
  - The highest priority task that is able to execute (is not blocked) is the task guaranteed by the RTOS to get processor time. The kernel will immediately suspend an executing task when a higher priority task becomes available.
  - Scheduling occurs automatically, with no explicit knowledge, structuring or commands within the application source code. But iIt is the responsibility of the application designers to ensure that tasks are allocated an appropriate priority.
  - When no task is able to execute the idle task will execute. The idle task has the option of placing the processor into power save mode.

- **Scheduler Configuration**
  - The scheduler is configured for preemptive operation. The kernel tick frequency should be set at the slowest value that provides the required time granularity.

- **Conclusion**
  - This can be a good solution provided the RAM and processing capacity is available. The partitioning of the application into tasks and the priority assigned to each task requires careful consideration.

Rozwiązanie 2:
Traditional Preemptive Multitasking System

- 🙂 Simple, segmented, flexible, maintainable design with few interdependencies.
- 🙂 Processor utilisation is automatically switched from task to task on a most urgent need basis with no explicit action required within the application source code.
- 😐 Power consumption can be reduced if the idle task places the processor into power save (sleep) mode, but may also be wasted as the tick interrupt will sometimes wake the processor unnecessarily.
- 😐 The kernel functionality will use processing resources. The extent of this will depend on the chosen kernel tick frequency.
- ☹ This solution requires a lot of tasks, each of which require their own stack, and many of which require a queue on which events can be received. This solution therefore uses a lot of RAM.
- ☹ Frequent context switching between tasks of the same priority will waste processor cycles.

Traditional Preemptive Multitasking System: Plant Control Task

```
#define CYCLE_RATE_MS 10
#define MAX_COMMS_DELAY 2

void PlantControlTask( void *pvParameters ) {
  portTickType xLastWakeTime;
  DataType Data1, Data2;
  // A
  InitialiseTheQueue();
  xLastWakeTime = xTaskGetTickCount();

  // B
  for( ;; ) {
    // C
    vTaskDelayUntil( &xLastWakeTime, CYCLE_RATE_MS );
    // Request data from the sensors.
    TransmitRequest();
    // D
    if( xQueueReceive( xFieldBusQueue, &Data1, MAX_COMMS_DELAY ) ) {
      // E
      if( xQueueReceive( xFieldBusQueue, &Data2, MAX_COMMS_DELAY ) ) {
        PerformControlAlgorithm();
        TransmitResults();
      }
    }
  }
}
```

- Implements all the control functionality: critical timing requirements therefore the highest priority within the system.

Traditional Preemptive Multitasking System: Embadded Web Server Task

- The embedded WEB server task can be represented by the following pseudo code. This only utilises processor time when data is available but will take a variable and relatively long time to complete. It is therefore given a low priority to prevent it adversely effecting the timing of the plant control, RS232 or keypad scanning tasks.

```
void WebServerTask( void *pvParameters ) {
  DataTypeA Data;
  for( ;; ) {
    // Block until data arrives. xEthernetQueue is filled by the
    // Ethernet interrupt service routine.
    if( xQueueReceive( xEthernetQueue, &Data, MAX_DELAY ) ) {
      ProcessHTTPData( Data );
    }
  }
}
```

Traditional Preemptive Multitasking System: RS232 Interface

- This is very similar in structure to the embedded WEB server task. It is given a medium priority to ensure it does not adversely effect the timing of the plant control task.

```
void RS232Task( void *pvParameters ) {
  DataTypeB Data;
  for( ;; ) {
    // Block until data arrives. xRS232Queue is filled by the
    // RS232 interrupt service routine.
    if( xQueueReceive( xRS232Queue, &Data, MAX_DELAY ) ) {
      ProcessSerialCharacters( Data );
    }
  }
}
```

Traditional Preemptive Multitasking System: Keypad Scanning Task

- It is given a medium priority as it's timing requirements are similar to the RS232 task.
- The cycle time is set much faster than the specified limit – it may not get processor time immediately upon request – and once executing may get pre-empted by the plant control task.

```
#define DELAY_PERIOD 4
void KeyScanTask( void *pvParmeters ) {
  char Key;
  portTickType xLastWakeTime;
  xLastWakeTime = xTaskGetTickCount();
  for( ;; ) {
    // Wait for the next cycle.
    vTaskDelayUntil( &xLastWakeTime, DELAY_PERIOD );

    // Scan the keyboard.
    if( KeyPressed( &Key ) ) {
      UpdateDisplay( Key );
    }
  }
}
```

Traditional Preemptive Multitasking System: Keypad Scanning Task

- If the overall system timing were such that this could be made the lowest priority task then the call to vTaskDelayUntil() could be removed altogether. The key scan function would then execute continuously whenever all the higher priority tasks were blocked - effectively taking the place of the idle task.

Traditional Preemptive Multitasking System: LED Task

- The simplest of all the tasks.

```
#define DELAY_PERIOD 1000
void LEDTask( void *pvParmeters ) {
  portTickType xLastWakeTime;
  xLastWakeTime = xTaskGetTickCount();
  for( ;; ) {
    // Wait for the next cycle.
    vTaskDelayUntil( &xLastWakeTime, DELAY_PERIOD );
    // Flash the appropriate LED.
    if( SystemIsHealthy() )
      FlashLED( GREEN );
    else
      FlashLED( RED );
  }
}
```

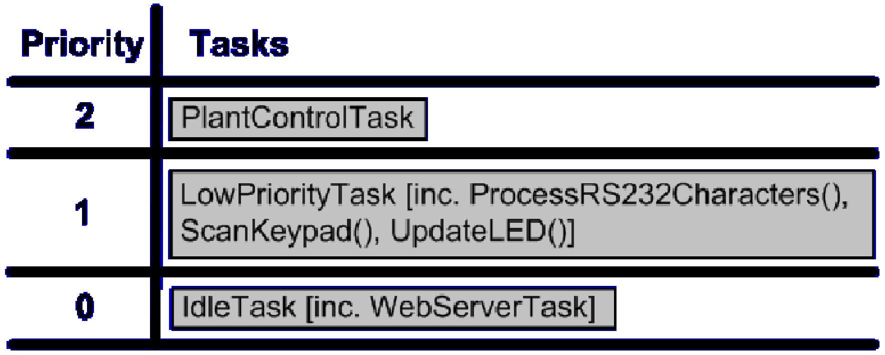
Rozwiązanie 2a:
Reducing RAM Utilisation

- Our hypothetical application can be split into three categories:
  - **Strict timing** - the plant control
    - A high priority task is created to service the critical control functionality.
  - **Deadline only timing** - the human interface
    - RS232, keyscan and LED functionality are grupped into a single medium priority task.
    - It is desirable for the embedded WEB server task to operate at a lower priority. Rather than creating a task specifically for the WEB server an idle task hook is implemented to add the WEB server functionality to the idle task. The WEB server must be written to ensure it never blocks!
  - **Flexible timing** - the LED
    - The LED functionality is too simple to warrant it's own task if RAM is at a premium. For reasons of demonstration this example includes the LED functionality in the single medium priority task. It could of coarse be implemented in a number of ways (from a peripheral timer for example).

- Tasks other than the idle task will block until an event indicates that processing is required. Events can either be external (ex. a key being pressed), or internal (ex. a timer expiring).

Rozwiązanie 2a: Reducing RAM Utilisation

- Concept of Operation
  - The grouping of functionality into the medium priority task has three important advantages over the infinite loop implementation presented in solution #2:
    - The use of a queue allows the medium priority task to block until an event causes data to be available - and then immediately jump to the relevant function to handle the event. This prevents wasted processor cycles - in contrast to the infinite loop implementation whereby an event will only be processed once the loop cycles to the appropriate handler.
    - The use of the real time kernel removes the requirement to explicitly consider the scheduling of the time critical task within the application source code.
    - The removal of the embedded WEB server function from the loop has made the execution time more predictable.
- Scheduler Configuration
  - The scheduler is configured for preemptive operation. The kernel tick frequency should be set at the slowest value that provides the required time granularity.
- Conclusion
  - This can be a good solution for systems with limited RAM but it is still processor intensive. Spare capacity within the system should be checked to allow for future expansion.



Priority	Tasks
2	PlantControlTask
1	LowPriorityTask [inc. ProcessRS232Characters(), ScanKeypad(), UpdateLED()]
0	IdleTask [inc. WebServerTask]

Rozwiązanie 2a:
Reducing RAM Utilisation

- 🙂 Creates only two application tasks so therefore uses much less RAM than solution #2.
- 🙂 Processor utilisation is automatically switched from task to task on a most urgent need basis.
- 🙂 Utilising the idle task effectively creates three application task priorities with the overhead of only two.
- 😐 The design is still simple but the execution time of the functions within the medium priority task could introduce timing issues. The separation of the embedded WEB server task reduces this risk and in any case any such issues would not effect the plant control task.
- 😐 Power consumption can be reduced if the idle task places the CPU into power save (sleep) mode, but may also be wasted as the tick interrupt will sometimes wake the CPU unnecessarily.
- 😐 The RTOS functionality will use processing resources. The extent of this will depend on the chosen kernel tick frequency.
- 🙁 The design might not scale if the application grows too large.

Rozwiązanie 2b: Reducing the Processor Overhead

- The critical plant control functionality is once again implemented by a high priority task but the use of the cooperative scheduler necessitates a change to its implementation.
  - Previously the timing was maintained using the vTaskDelayUntil() API function. When the preemptive scheduler was used, assigning the control task the highest priority ensured it started executing at exactly the specified time.
  - Now the cooperative scheduler is being used - therefore a task switch will only occur when explicitly requested from the application source code so the guaranteed timing is lost.

- Solution #4 uses an interrupt from a peripheral timer to ensure a context switch is requested at the exact frequency required by the control task. The scheduler ensures that each requested context switch results in a switch to the highest priority task that is able to run.
- The keypad scanning function also requires regular processor time so it too is executed within the task triggered by the timer interrupt. The timing of this task can be easily evaluated; The worst case processing time of the control function is given by the error case - when no data is forthcoming from the networked sensors causing the control function to time out. The execution time of the keypad scanning function is basically fixed. We can therefore be certain that chaining their functionality in this manner will never result in jitter in the control cycle frequency - or worse still a missed control cycle.
- The RS232 task will be scheduled by the RS232 interrupt service routine.
- The flexible timing requirements of the LED functionality means it can probably join the embedded WEB server task within the idle task hook. If this is not adequate then it too can be moved up to the high priority task.

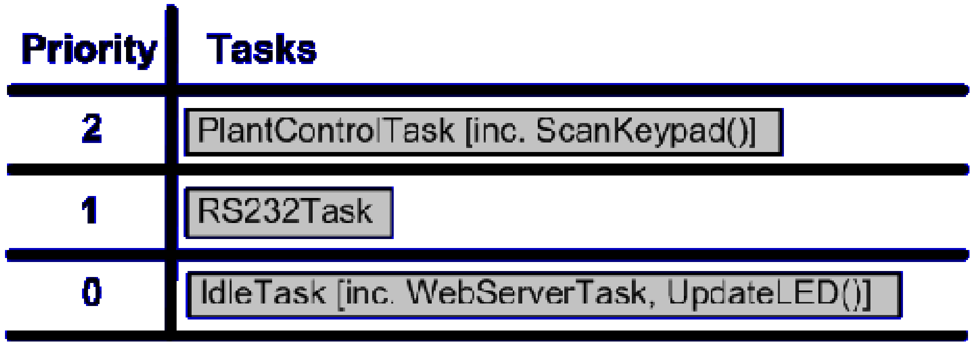
Rozwiązanie 2b: Reducing the Processor Overhead

- **Concept of Operation**
  - The cooperative scheduler will only perform a context switch when one is explicitly requested. This greatly reduces the processor overhead imposed by the RTOS. The idle task, including the embedded WEB server functionality, will execute without any unnecessary interruptions from the kernel. An interrupt originating from either the RS232 or timer peripheral will result in a context switch exactly and only when one is necessary. This way the RS232 task will still pre-empt the idle task, and can still itself be pre-empted by the plant control task - maintaining the prioritised system functionality.

Scheduler Configuration

  - The scheduler is configured for cooperative operation. The kernel tick is used to maintain the real time tick value only.

- **Conclusion**
  - Features of the RTOS kernel can be used with very little overhead, enabling a simplified design even on systems where processor and memory constraints prevent a fully preemptive solution.



Priority	Tasks
2	PlantControlTask [inc. ScanKeypad()]
1	RS232Task
0	IdleTask [inc. WebServerTask, UpdateLED()]

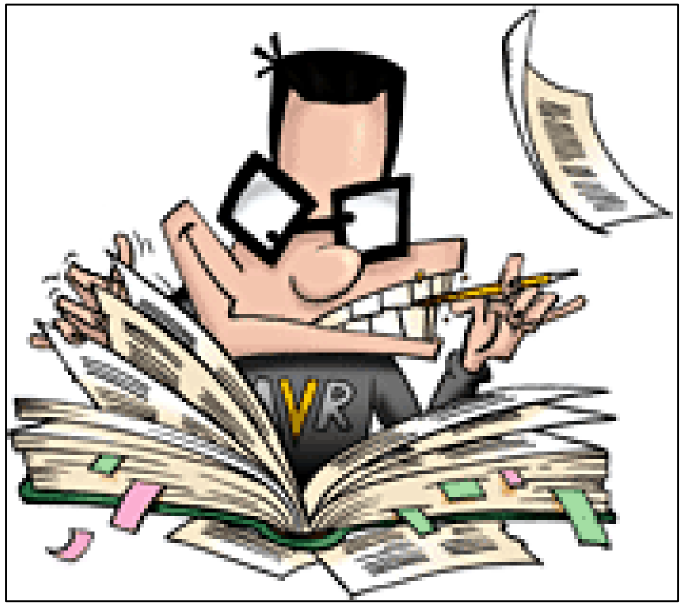
Rozwiązanie 2b: Reducing the Processor Overhead

- 🙂 Creates only two application tasks so therefore uses much less RAM than solution #2.
- 🙂 The RTOS processing overhead is reduced to a minimum.
- 😐 Only a subset of the RTOS features are used. This necessitates a greater consideration of the timing and execution environment at the application source code level, but still allows for a greatly simplified design (when compared to solution #1).
- 🙁 Reliance on processor peripherals. Non portable.
- 🙁 The problems of analysis and interdependencies between modules as were identified with solution #1 are starting to become a consideration again - although to a much lesser extent.
- 🙁 The design might not scale if the application grows too large

Przykładowe systemy RTOS

- FreeRTOS
  - http://www.freertos.org/
- uC/OS-II
  - http://www.micrium.com/
- pC/OS
  - http://www.embedded-os.de/
- Ethernut
  - http://www.ethernut.de/
- Contiki - The Operating System for Embedded Smart Objects - the Internet of Things
  - http://www.sics.se/contiki/
- AvrX Real Time Kernel
  - http://www.barello.net/avrx/index.htm
- uClinux - Embedded Linux/Microcontroller Project
  - http://uclinux.org/

Pytania?



Pytania?

Politechnika Łódzka

Instytut Elektroniki

Dziękuję za uwagę.