CENG 314
Embedded Computer Systems
Lecture Notes

Real-Time Operating Systems
for
Microcontrollers

Asst. Prof. Tolga Ayav, Ph.D.

Department of Computer Engineering
İzmir Institute of Technology

Real-Time Systems

It can be argued that
all practical systems are real-time!

Hard Real-Time

Systems where failure to meet system response time constraints leads to a system failure are called hard real-time systems.

Soft Real-Time:

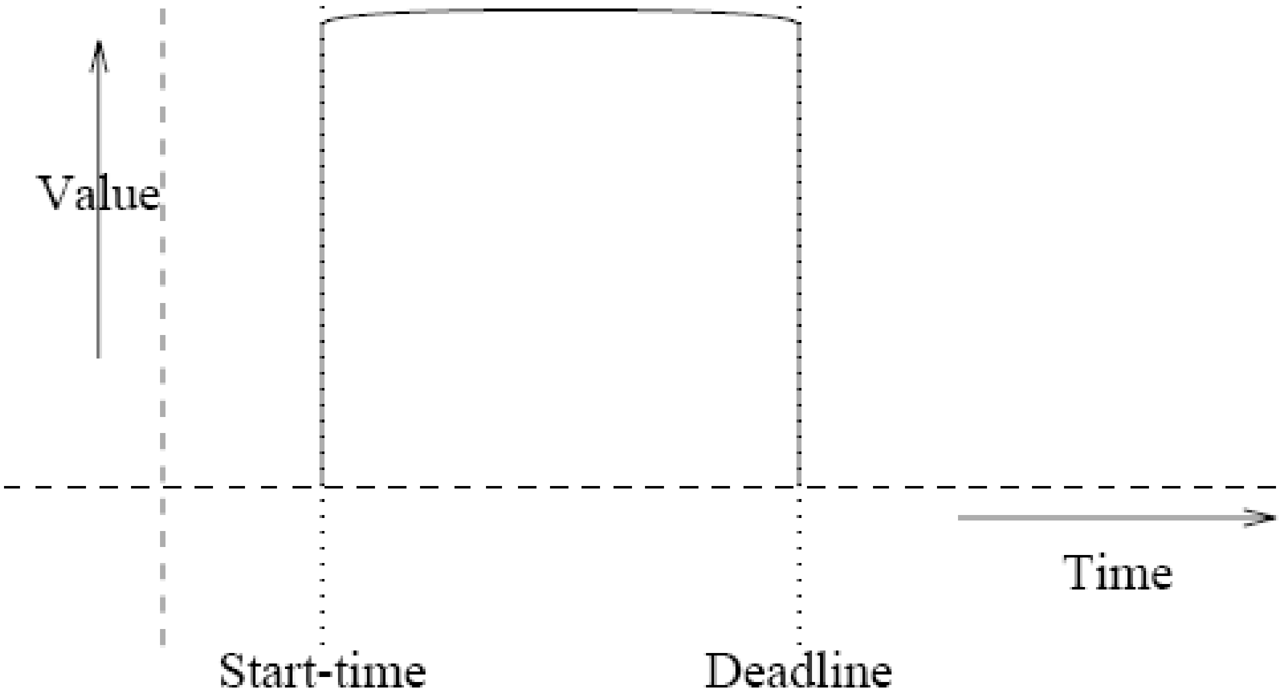
Systems where performance is degraded but not destroyed by failure to meet system response time constraints.

Firm Real-Time:

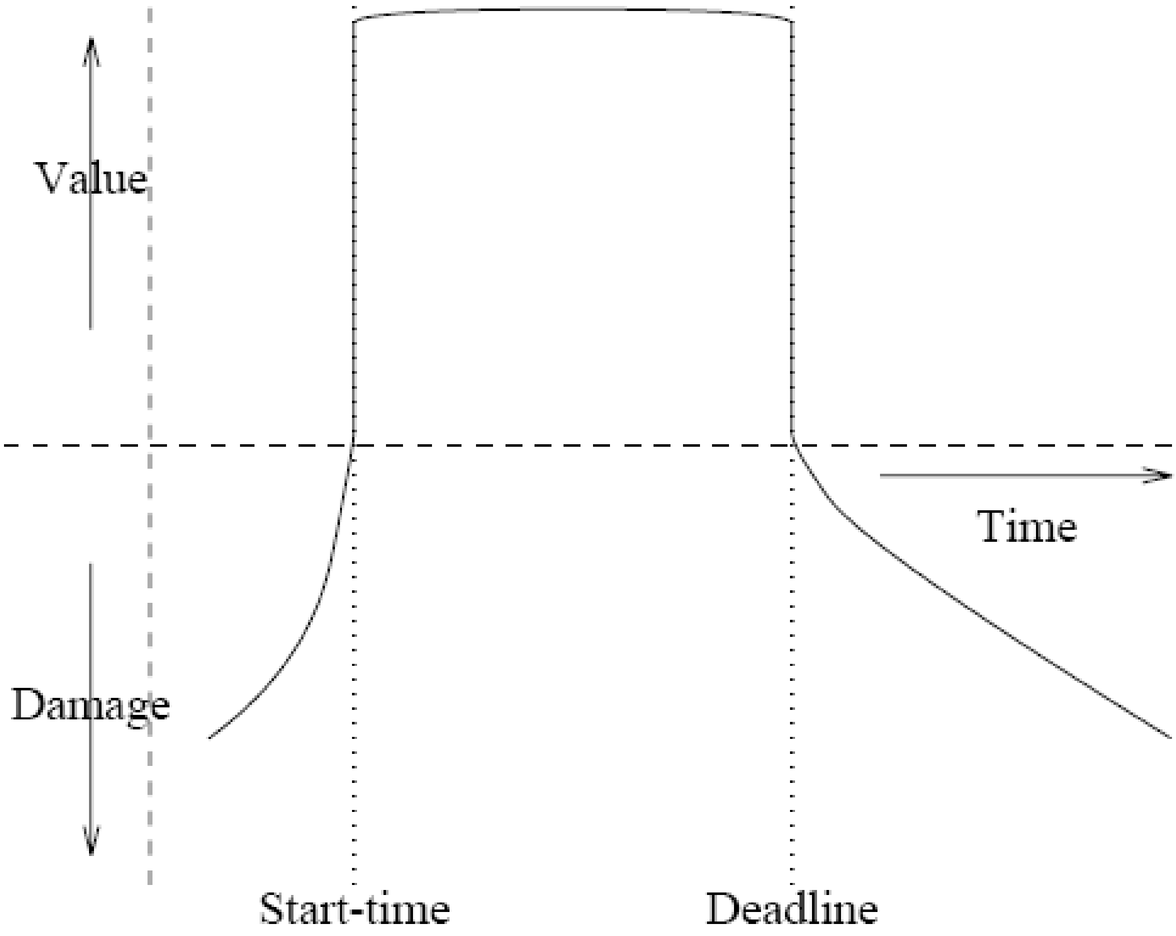
Systems with hard deadlines where some low probability of missing deadline can be tolerated.

Task Characteristics in terms of System Requirements

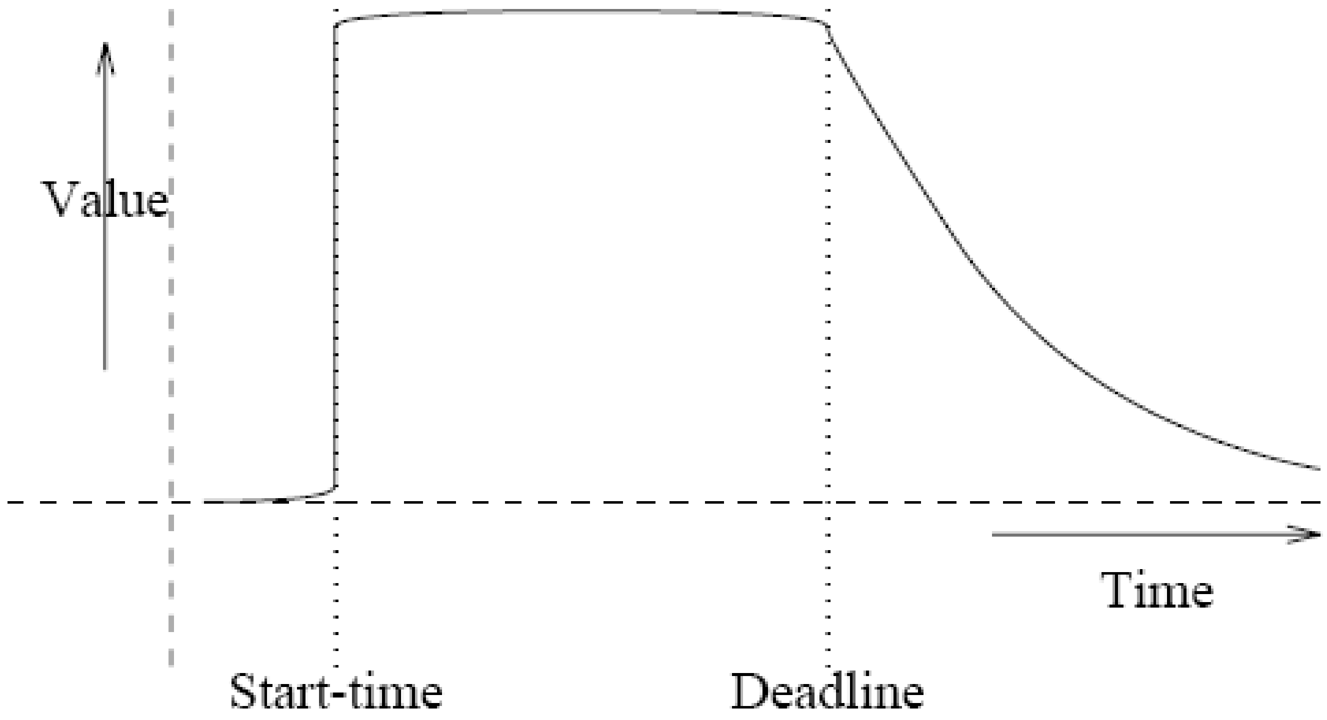
Hard Deadline



Safety Critical System



Soft Deadline



Real-Time Kernels

- All operating systems must provide three specific functions:
  - Task management
  - Task scheduling
  - Intertask communication
- A Kernel, executive or nucleus is the smallest portion of the OS that provides these functions
- Rea-Time kernels must provide:
  - A. Interrupt handling, guaranteed interrupt response
  - B. Process management (Support for scheduling of real-time processes and preemptive scheduling)
  - C. Interprocess communication and synchronization.
  - D. Time management.
  - E. Memory management
  - F. I/O support (Support for communication with peripheral devices via drivers)
  - G. High speed data acquisition
  - H. Resource management (User control of system resources)
  - I. Error and exception handling

Real-Time Kernel Features

- A real-time OS should provide support for the creation, deletion and scheduling of multiple processes
- A real-time OS must be able to response an event and take deterministic (well-defined in terms of function and time)
- action based on the event.
- A real-time OS should support interprocess communications via reliable and fast facilities, such as semaphores, shared memory and message passing.
- A real-time system must be able to handle very high burst rates in high speed data acquisition applications.

RT Scheduling

- Among many functions, scheduling is the most important function of a real-time kernel

- A realtime application is composed as a set of coordinated tasks. We can categorize the task according to their activation:
  - Periodic tasks
  - Sporadic tasks
  - Aperiodic tasks

  -

- Periodic tasks are started at regular intervals and has to be completed before some deadline.
- Sporadic tasks are appeared irregularly, but within a bounded frequency.
- Aperiodic tasks’ parameters are completely unknown.

RT Tasks

- We can use the following quintuple to express task τi:
  - $< \tau_i, b_i, c_i, f_i, d_i>$

- b_i is begin time of τ_i
- c_i is computation time of τ_i
- d_i is the deadline
- f_i is the frequency (for sporadic tasks it's the bound)

- For schedulability, at least the following conditions must be met:
  - $c_i < d_i - b_i < f_i$
  - $\sum c_i f_i \leq$ available resource

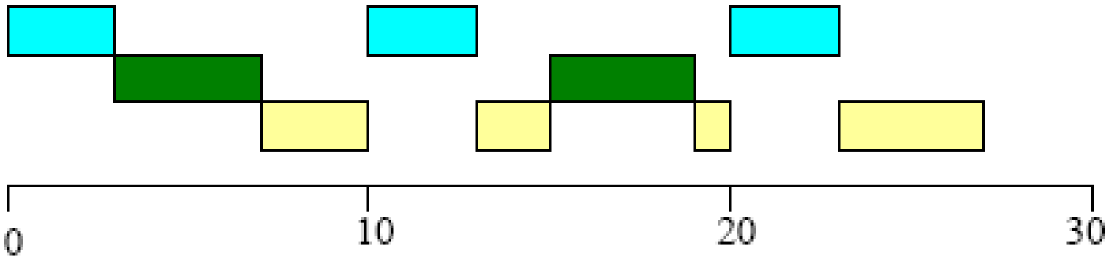
RT Tasks

- We can also categorize tasks according to their time criticality:
  - Hard real-time tasks
  - Soft real-time tasks
  - Non real-time tasks (background tasks)

Task Scheduling

- Three periodic tasks: A, B, C
- T is period, D is deadline and C is execution time
- uniprocessor

Task	T	D	C
A	10	10	3
B	15	15	4
C	30	30	10



Scheduling Techniques

- Dynamic Scheduling
  - Static priority-driven preemptive scheduling(RM)
  - Dynamic priority-driven preemptive scheduling(EDF)
  - Adaptive scheduling(FC-EDF)
  - ...
- Static Scheduling
  - AAA (algorithm architecture adequation)
  - ...

Rate-Monotonic Scheduling

Assumptions:

1. Simple task model: No interprocess communication and all tasks are periodic
2. Tasks have priorities which are inversly proportional to their periods.
3. Tasks' deadlines are equal to their periods.
4. A high priority task may preempt lower priority tasks.

Liu and Layland (1973) proved that for a set of *n* periodic tasks with unique periods, a feasible schedule that will always meet deadlines exists if the CPU utilization is:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(\sqrt[n]{2} - 1)$$

Where C_i is the computation time of a task i, T_i is the deadline of task i and *n* is the number of tasks.

For example, for n=2, $U \leq 0.8284$

Rate-Monotonic Scheduling

When number of tasks approaches to infinity, this utilization bound will converge to:

$$\lim_{n \to \infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.693147\ldots$$

Example:

Task	Execution Time	Period
τ1	1	8
τ2	2	5
τ3	2	10

$$\frac{1}{8} + \frac{2}{5} + \frac{2}{10} = 0.725$$

$$U = 3(2^{\frac{1}{3}} - 1) = 0.77976\ldots$$

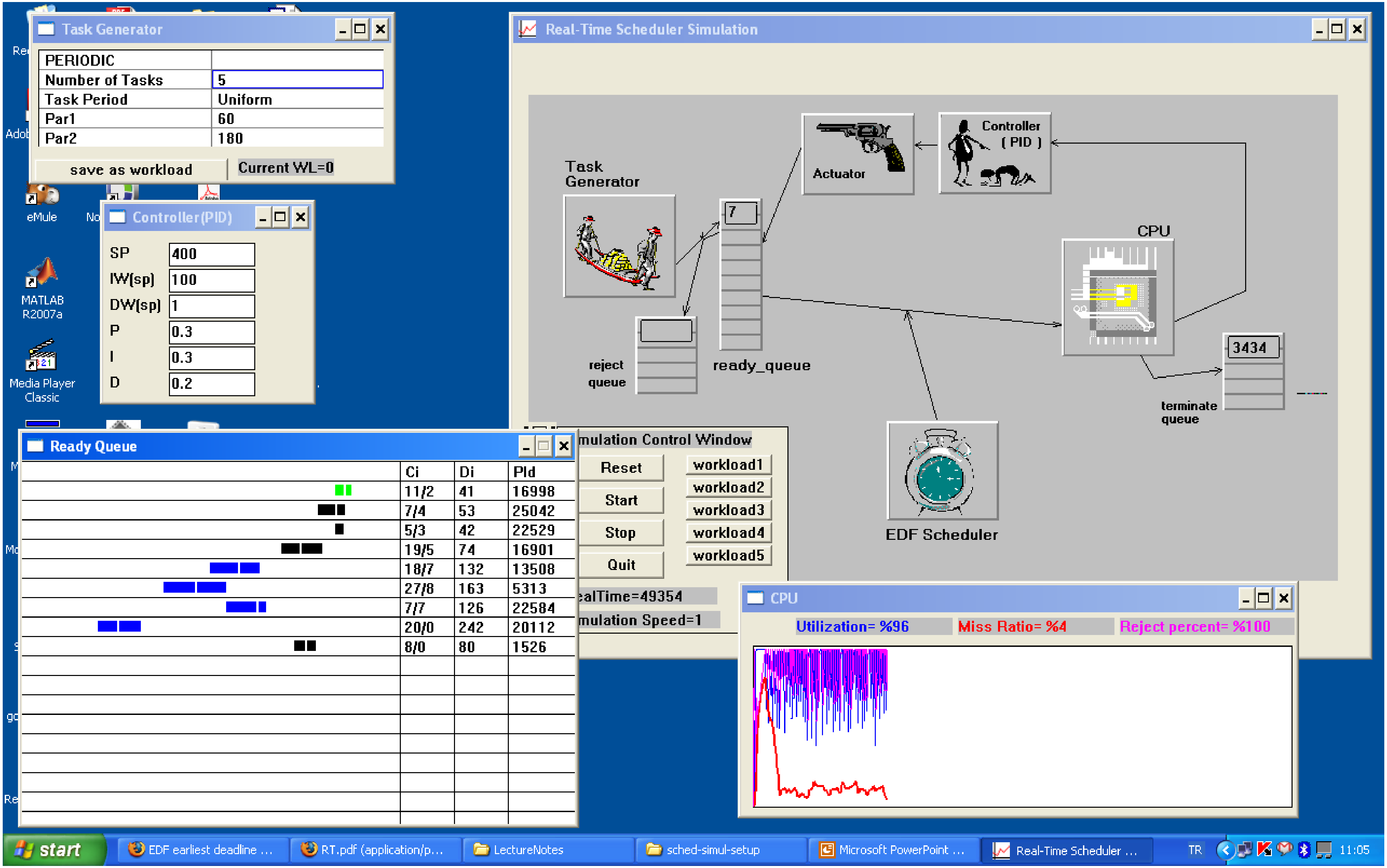
$$0.725 < 0.77976\ldots$$

Thus, the system is schedulable

Earliest-Deadline-First Scheduling

- Priorities are changed dynamically
- Task with the earliest deadline gets the highest priority
- Unless RM, utilization may go up to 100%

FC-EDF Scheduler Simulator



PERIODIC	
Number of Tasks	5
Task Period	Uniform
Par1	60
Par2	180

SP	400
IW(sp)	100
DW(sp)	1
P	0.3
I	0.3
D	0.2

	Ci	Di	Pld
	11/2	41	16998
	7/4	53	25042
	5/3	42	22529
	19/5	74	16901
	18/7	132	13508
	27/8	163	5313
	7/7	126	22584
	20/0	242	20112
	8/0	80	1526

ealTime=49354

mulation Speed=1

Utilization= %96 Miss Ratio= %4 Reject percent= %100

RT-Linux

- **RT-Linux is an operating system, in which a small real-time kernel co-exists with standard Linux kernel**

– The real-time kernel sits between *standard Linux kernel* and the *h/w*.

– The standard Linux kernel sees this real-time layer as actual h/w

– The real-time kernel *intercepts all hardware interrupts*.

  • Only for those RTLinux-related interrupts, the appropriate ISR is run.

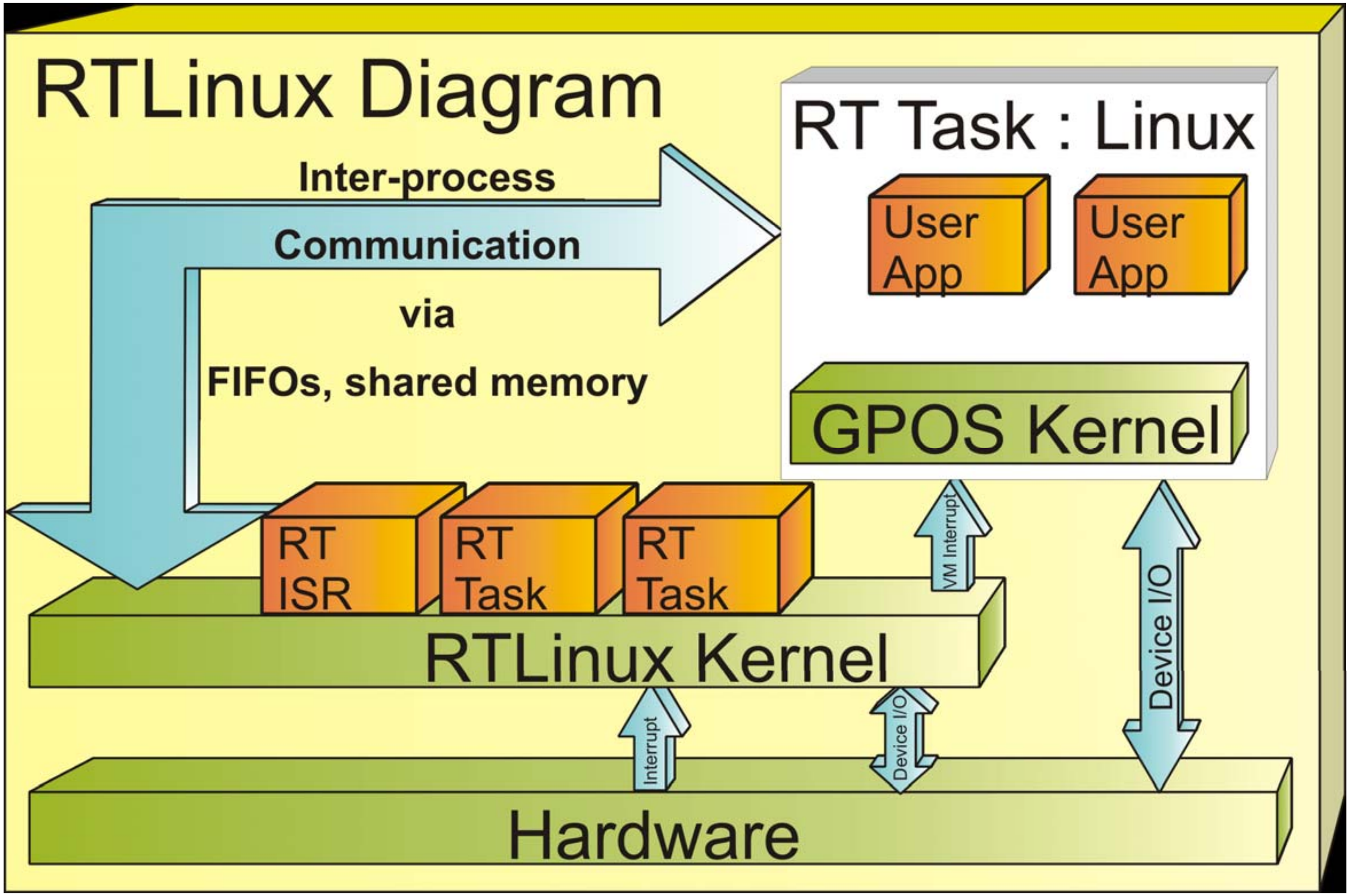
  • All other interrupts are held and passed to the standard Linux kernel as software interrupts when the standard Linux kernel runs.

– The real-time kernel assigns the *lowest priority* to the *standard Linux kernel*. Thus the realtime tasks will be executed in real-time

– user can create realtime tasks and achieve correct timing for them by deciding on scheduling algorithms, priorities, execution freq, etc.

– Realtime tasks are *privileged* (that is, they have direct access to hardware), and they do *NOT use virtual memory*.

RT-Linux


RTLinux Diagram
Inter-process
Communication
via
FIFOs, shared memory
RT Task : Linux
User App
User App
GPOS Kernel
RT ISR
RT Task
RT Task
VM Interrupt
Device I/O
RTLinux Kernel
Interrupt
Device I/O
Hardware

Scheduler

- **RT-Linux contains a dynamic scheduler**
- **RT-Linux has many kinds of schedulers**
  – The EDF (Earliest Deadline First) scheduler
  – Rate-monotonic scheduler
- **Real-time FIFOs**
  – RT-FIFOs are used to pass information between real-time process and ordinary Linux process.
  – RT-FIFOs are designed to never block the real-time tasks.
  – RT-FIFOs are, like realtime tasks, never page out. This eliminates the problem of unpredictable delay due to paging.

Linux v.s. RTLinux

- **Linux Non-real-time Features**

– Linux scheduling algorithms are not designed for real-time tasks

- Provide good *average* performance or throughput

– Unpredictable delay

- Uninterruptible system calls, the use of interrupt disabling virtual memory support (context switch may take hundreds of microsecond).

  – Linux Timer resolution is coarse, 10ms

  – Linux Kernel is Non-preemptible.

- **RTLinux Real-time Features**

– Support real-time scheduling

– Predictable delay (by its small size and limited operations)

– Finer time resolution

– Preemptible kernel

– No virtual memory support

Kernels for Microcontrollers

RTX 51 Tiny Real-Time Kernel

Why C is common?

- It’s a mid-level language with high-level features (functions and modules) and low-level features (hardware access via pointers)
- It’s very efficient
- It’s popular and well-understood
- C syntax is easy
- Good, well-proven compilers are available for every embedded processor
- Experienced staff are available
- Books, courses, code samples and WWW sites are widely available

The super-loop software architecture

Problem

What is the minimum software environment you need to create an embedded C program?

Solution

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

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

Crucially, the ‘super loop’, or ‘endless loop’, is required because we have no operating system to return to: our application will keep looping until the system power is removed.

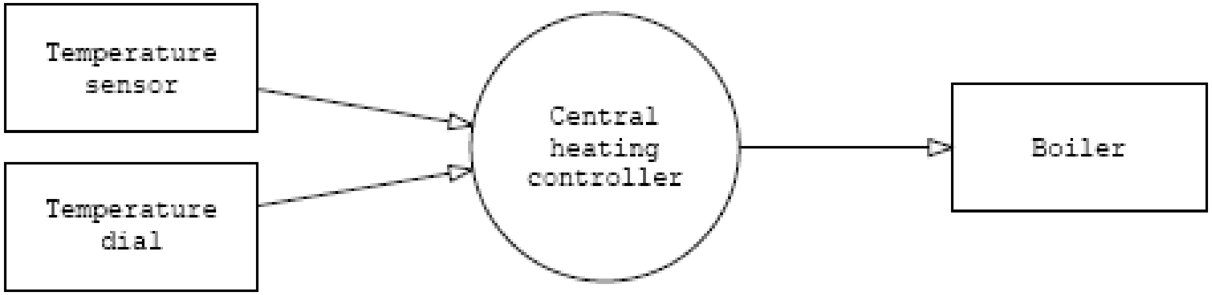
Strengths and weaknesseses of “super loops”

- ☺ The main strength of Super Loop systems is their simplicity. This makes them (comparatively) easy to build, debug, test and maintain.
- ☺ Super Loops are highly efficient: they have minimal hardware resource implications.
- ☺ Super Loops are highly portable.

BUT:

- ☹ If your application requires accurate timing (for example, you need to acquire data precisely every 2 ms), then this framework will not provide the accuracy or flexibility you require.
- ☹ The basic Super Loop operates at ‘full power’ (normal operating mode) at all times. This may not be necessary in all applications, and can have a dramatic impact on system power consumption.

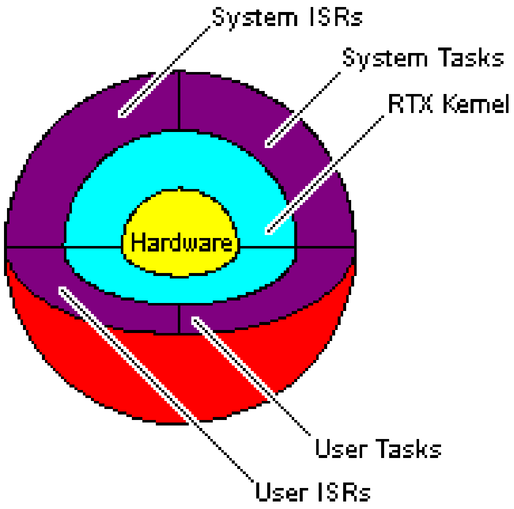
Example: Central-heating controller



```
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();
      }
   }
```

RTX 51 Tiny RT Kernel

RTX51 Tiny is a small real-time kernel designed for single-chip applications where code size is the most important factor. The RTX51 Tiny kernel requires only 900 bytes of code space and is well-suited for applications that don't need RTOS features like messaging, semaphores, and memory pool management.



- RTX51 Tiny was designed for single-chip applications where no **XDATA** is available. However, RTX51 Tiny may be used with any 8051 target system.
- RTX51 Tiny supports all memory models of the C51 Compiler (SMALL, COMPACT, and LARGE). Operating system variables and task stacks are stored in internal DATA/IDATA memory.
- RTX51 Tiny performs round-robin and cooperative multitasking only. Preemptive task switching and task priorities are not supported. If you need these features, you should consider RTX51
- RTX51 Tiny uses Timer 0 for the operating system timer tick. No other hardware resources are used.
- RTX51 Tiny is royalty-free.

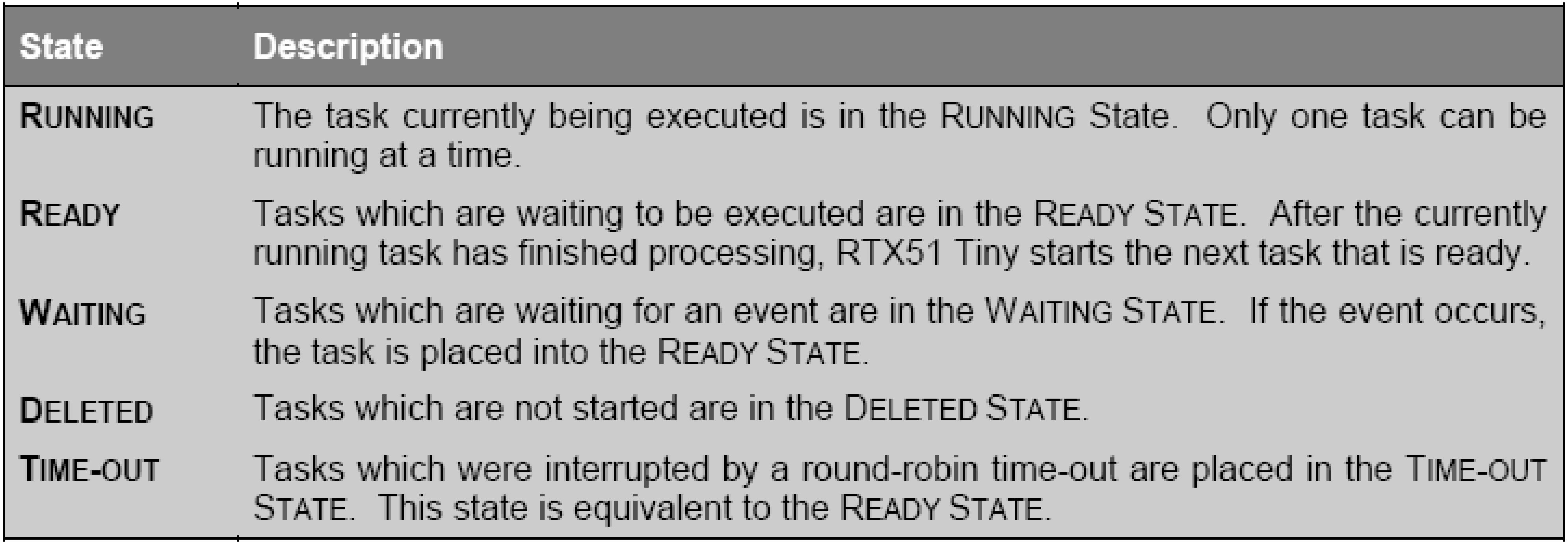
Task Definition

```
void func (void) _task_ num
```

where ***num*** is a task ID number from 0 to 15.

```
void job0 (void) _task_ 0 {
  while (1) {
   counter0++; /* increment counter */
  }
}
```

Task States

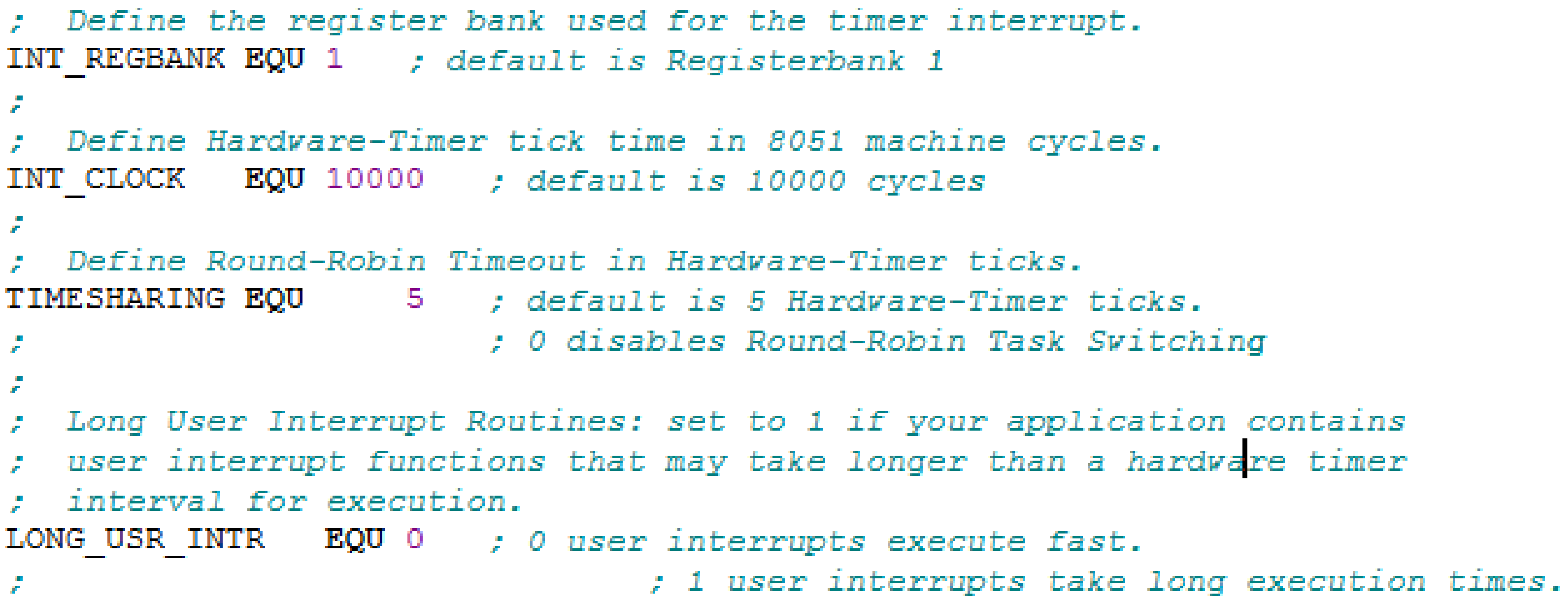


State	Description
RUNNING	The task currently being executed is in the RUNNING State. Only one task can be running at a time.
READY	Tasks which are waiting to be executed are in the READY STATE. After the currently running task has finished processing, RTX51 Tiny starts the next task that is ready.
WAITING	Tasks which are waiting for an event are in the WAITING STATE. If the event occurs, the task is placed into the READY STATE.
DELETED	Tasks which are not started are in the DELETED STATE.
TIME-OUT	Tasks which were interrupted by a round-robin time-out are placed in the TIME-OUT STATE. This state is equivalent to the READY STATE.

Task Switching

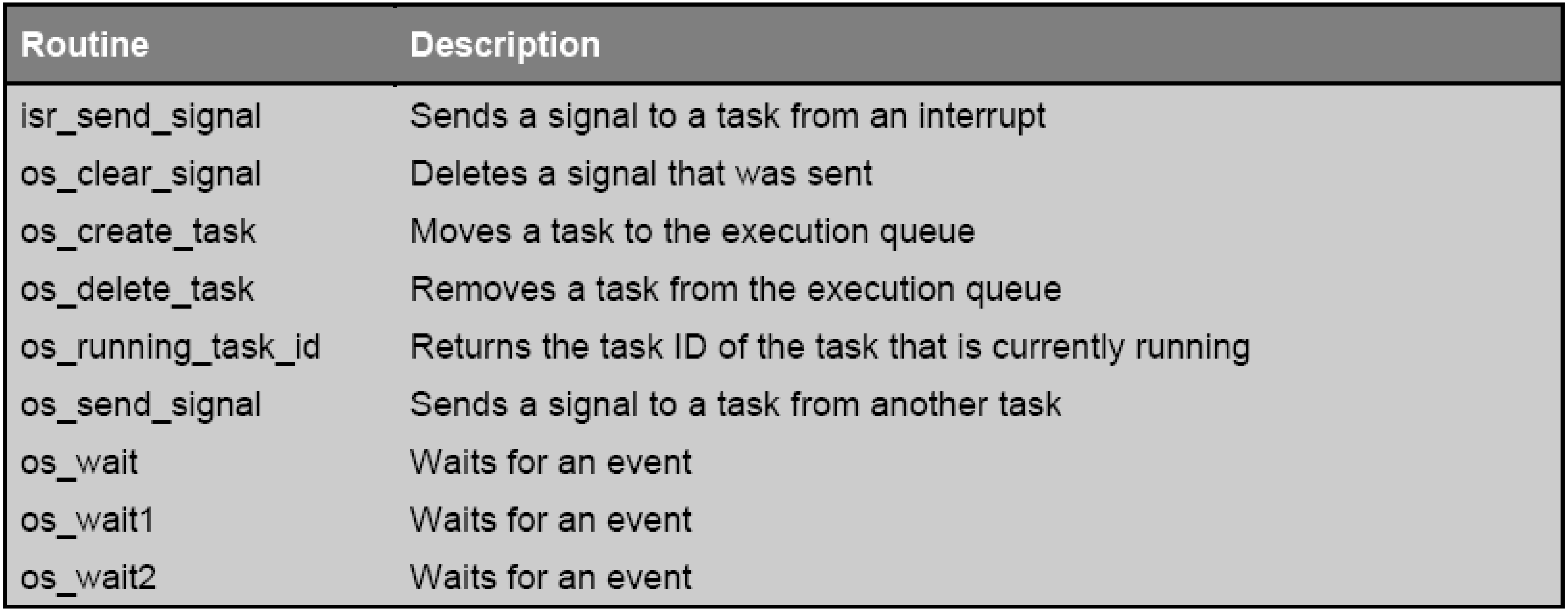
- RTX51 Tiny performs Round-Robin Scheduling
- CPU time is divided into time slices
- The duration of a time slice can be defined with the configurarion variable**TIMESHARING**.
- Rather then wait for a task’s time slice to expire, you can use the **os_wait** system function to signal RTX51

CONF_TNY.A51



```
;  Define the register bank used for the timer interrupt.
INT_REGBANK EQU 1    ; default is Registerbank 1
;
;  Define Hardware-Timer tick time in 8051 machine cycles.
INT_CLOCK   EQU 10000    ; default is 10000 cycles
;
;  Define Round-Robin Timeout in Hardware-Timer ticks.
TIMESHARING EQU     5    ; default is 5 Hardware-Timer ticks.
;                        ; 0 disables Round-Robin Task Switching
;
;  Long User Interrupt Routines: set to 1 if your application contains
;  user interrupt functions that may take longer than a hardware timer
;  interval for execution.
LONG_USR_INTR   EQU 0    ; 0 user interrupts execute fast.
;                                ; 1 user interrupts take long execution times.
```

RTX51 Tiny System Functions



Routine	Description
isr_send_signal	Sends a signal to a task from an interrupt
os_clear_signal	Deletes a signal that was sent
os_create_task	Moves a task to the execution queue
os_delete_task	Removes a task from the execution queue
os_running_task_id	Returns the task ID of the task that is currently running
os_send_signal	Sends a signal to a task from another task
os_wait	Waits for an event
os_wait1	Waits for an event
os_wait2	Waits for an event

isr_send_signal

char isr_send_signal (**unsigned char** *task_id*);

The **isr_send_signal** function sends a signal to task *task_id*. If the specified task is already waiting for a signal, this function call will ready the task for execution. Otherwise, the signal is stored in the signal flag of the task.The **isr_send_signal** function may be called only from interrupt functions.

```
#include <rtx51tny.h>

void tst_isr_send_signal (void) interrupt 2
{
  .
  .
   isr_send_signal (8); /* signal task #8 */
  .
  .
  .
}
```

os_create_task

```
char os_create_task (unsigned char task_id);
```

The **os_create_task** function starts the defined task function using the task number specified by *task_id*. The task is marked as ready and is executed according to the rules specified for RTX51 Tiny.

```
#include <rtx51tny.h>
#include <stdio.h> /* for printf */
void new_task (void) _task_ 2
{
   ...
}

void tst_os_create_task (void) _task_ 0
{
  .
  if (os_create_task (2)){
    printf ("Couldn't start task 2\n");
  }
  .
  .
}
```

os_delete_task

```
char os_delete_task (unsigned char task_id);
```

The **os_delete_task** function stops the task specified by the *task_id* argument. The specified task is removed from the task list.

```
#include <rtx51tny.h>
#include <stdio.h> /* for printf */

void tst_os_delete_task (void) _task_ 0
{
  .
  if (os_delete_task (2)){
    printf ("Couldn't stop task 2\n");
  }
  .
  .
}
```

os_delete_task

```
char os_delete_task (unsigned char task_id);
```

The **os_delete_task** function stops the task specified by the *task_id* argument. The specified task is removed from the task list.

```
#include <rtx51tny.h>
#include <stdio.h> /* for printf */


void tst_os_delete_task (void) _task_ 0
{
  .
  if (os_delete_task (2)){
    printf ("Couldn't stop task 2\n");
  }
  .
  .
}
```

os_running_task_id

```
char os_running_task_id (void);
```

The **os_running_task_id** function determines the task id of the currently executing task function.

```
#include <rtx51tny.h>
#include <stdio.h> /* for printf */

void tst_os_running_task (void) _task_ 3
{
  unsigned char tid;
  tid = os_running_task_id ();
  /* tid = 3 */
}
```

os_running_task_id

```
char os_running_task_id (void);
```

The **os_running_task_id** function determines the task id of the currently executing task function.

```
#include <rtx51tny.h>
#include <stdio.h> /* for printf */

void tst_os_running_task (void) _task_ 3
{
   unsigned char tid;
   tid = os_running_task_id ();
   /* tid = 3 */
}
```

os_send_signal

```
char os_send_signal ( unsigned char task_id);
```

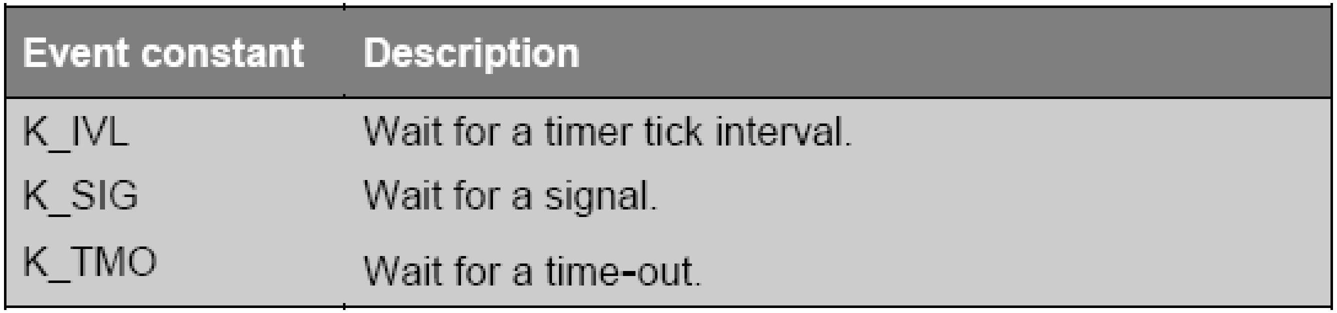
The **os_send_signal** function sends a signal to task *task_id*. If the specified task is already waiting for a signal, this function call readies the task for execution. Otherwise, the signal is stored in the signal flag of the task.
The **os_send_signal** function may be called only from task functions.

```
#include <rtx51tny.h>
#include <stdio.h> /* for printf */
void signal_func (void) _task_ 2
{
   ...
   os_send_signal (8); /* signal task #8 */
   ...
}

void tst_os_send_signal (void) _task_ 8
{
   ...
   os_send_signal (2); /* signal task #2 */
   ...
}
```

os_wait

char os_wait (**unsigned char** *event_sel*, **unsigned char** *ticks*, **unsigned int** *dummy*);

The **os_wait** function halts the current task and waits for one or several events such as a time interval, a time-out, or a signal from another task or interrupt. The *event_sel* argument specifies the event or events to wait for and can be any combination of the following manifest constants:



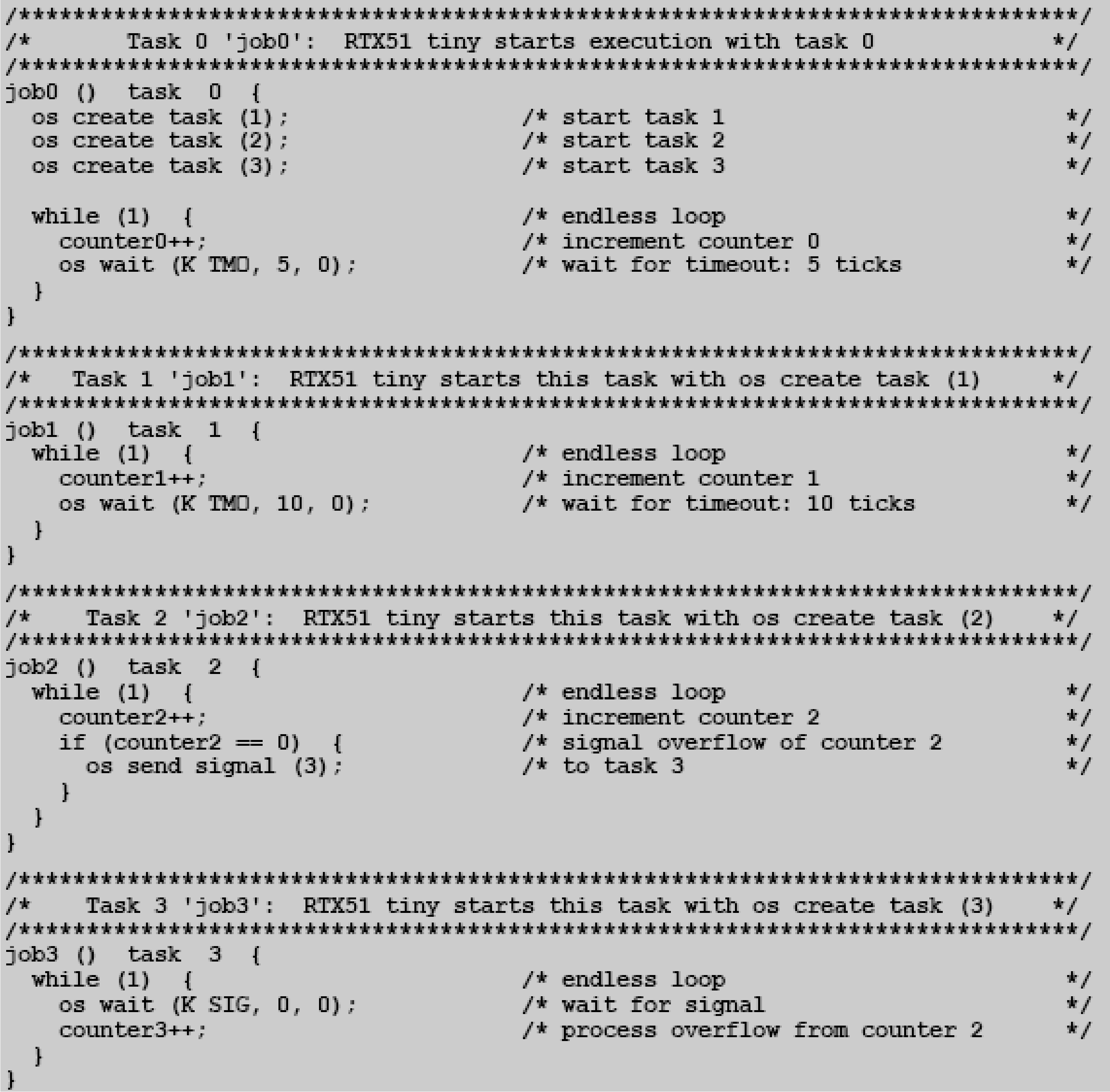
Event constant	Description
K_IVL	Wait for a timer tick interval.
K_SIG	Wait for a signal.
K_TMO	Wait for a time-out.

K_TMO | K_SIG, specifies that the task wait for a time-out or for a signal.
The t*icks* argument specifies the number of timer ticks to wait for an interval event (**K_IVL**) or a time-out event (**K_TMO**).

os_wait example

```
void tst_os_wait (void) _task_ 9
{
 char event;
   while (1) {
      event = os_wait (K_SIG + K_TMO, 50, 0);
      switch (event){
         default:
         /* this should never happen */
         break;
         case TMO_EVENT: /* time-out */
         /* 50 tick time-out occurred */
         break;

         case SIG_EVENT: /* signal recvd */
         /* signal received */
         break;
      }
   }
}
```

Example Application



```
/******************************************************************************/
/*       Task 0 'job0':  RTX51 tiny starts execution with task 0             */
/******************************************************************************/
job0 ()  task  0  {
  os create task (1);                 /* start task 1                         */
  os create task (2);                 /* start task 2                         */
  os create task (3);                 /* start task 3                         */

  while (1)  {                        /* endless loop                         */
    counter0++;                       /* increment counter 0                  */
    os wait (K TMO, 5, 0);            /* wait for timeout: 5 ticks            */
  }
}

/******************************************************************************/
/*   Task 1 'job1':  RTX51 tiny starts this task with os create task (1)     */
/******************************************************************************/
job1 ()  task  1  {
  while (1)  {                        /* endless loop                         */
    counter1++;                       /* increment counter 1                  */
    os wait (K TMO, 10, 0);           /* wait for timeout: 10 ticks           */
  }
}

/******************************************************************************/
/*    Task 2 'job2':  RTX51 tiny starts this task with os create task (2)    */
/******************************************************************************/
job2 ()  task  2  {
  while (1)  {                        /* endless loop                         */
    counter2++;                       /* increment counter 2                  */
    if (counter2 == 0)  {             /* signal overflow of counter 2         */
      os send signal (3);             /* to task 3                            */
    }
  }
}

/******************************************************************************/
/*    Task 3 'job3':  RTX51 tiny starts this task with os create task (3)    */
/******************************************************************************/
job3 ()  task  3  {
  while (1)  {                        /* endless loop                         */
    os wait (K SIG, 0, 0);            /* wait for signal                      */
    counter3++;                       /* process overflow from counter 2      */
  }
}
```