Designing Responsive and Real-Time Systems

Chapter 10

Renesas Electronics America Inc.
Embedded Systems using the RX63N

Rev. 1.0

Learning Objectives

- Most Embedded Systems have multiple real time tasks running at the same time.
- “Which task should it perform first?” determines how responsive the Processor is.
- Response of a Processor determines many important matrices while choosing the right processor for our application.

- In this chapter we will see how to design Responsive and Real time Embedded Systems along with some task scheduling approaches.

Scheduling

- Topic—How do we make the processor do things at the right times?
- There are various methods, the best fit depends on:
  - System requirements: response time
  - Software complexity: number of threads of execution
  - Resources: RAM, interrupts, energy available

Scheduling

- How do we schedule the tasks on the CPU?
  - An infinite loop in main.
  - Real-time operating system.
  - Is there anything else available?

Scheduling

- A Simple Scheduling Example
  - Consider an embedded system which controls a doorbell in a house. When someone presses a switch, the doorbell should ring. The system’s **responsiveness** describes how long it takes from pressing the switch to sounding the bell.
    - Simple code to realize the above example:

```
void main (void){
   init_system();
   while(1){
      if(switch == PRESSED){
         Ring_The_Bell();
      }
   }
}
```

Scheduling

- The doorbell in the previous example is very responsive. What if we decide to throw in burglar detector and a smoke detector?
    - Our while loop should look like this:

```
while(1){
   if(switch == PRESSED){
      Ring_The_Doorbell();
   }
   if(Burglar_Detected() == TRUE){
      Sound_The_Burglar_Alarm();
   }
   if(Smoke_Detected() == TRUE){
      Sound_The_Fire_Alarm();
   }
}
```

Scheduling

- How should we share the processor’s time between these tasks?

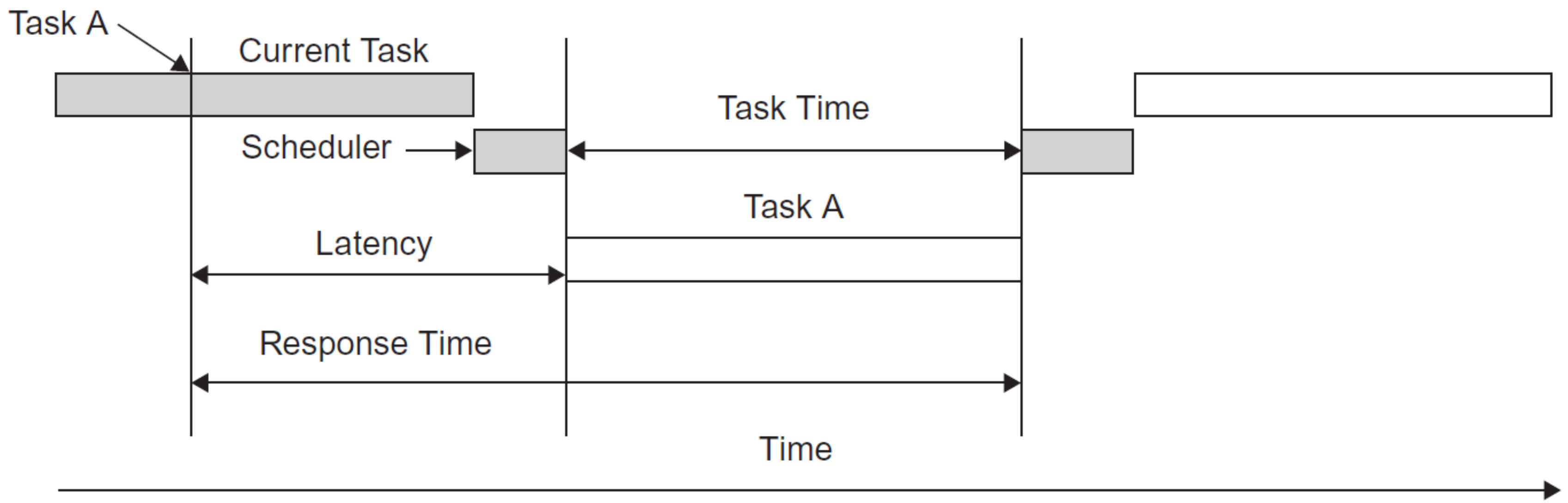
- It depends on various factors:
  - Delay between smoke detection and alarm going off?
  - What if smoke is detected while someone is at the door and someone is trying to break thorough the window? What gets precedence?

- Now that we have to share the processor, we have to worry about how long the bell rings and the alarms sound. If doorbell rings for 30 seconds, it will run for 30 seconds. During this time, we won’t know if our house is burning or being robbed.

Scheduling

- This example reveals the two fundamental issues in scheduling for responsive system:
  - If we have multiple tasks ready to run, which one do we run first? This decision defines the **ordering** of task execution.
  - Do we allow one task to interrupt or **preempt** another task (or even itself)?
- Both of these decisions will determine the system's **responsiveness** (measured by response time).

Task Ordering



- $T_{release}$ (i) = Time at which task i becomes ready to run.
- $T_{response}$ (i) = Delay between request for service and completion of service for task i.
- T_{task} (i) = Time needed to perform computations for task i.
- T_{ISR}(i) = Time needed to perform interrupt service routine i.

Going Dynamic

- We can change the order based on current conditions (e.g., if the house is on fire) using a **dynamic** schedule.
- We will try to take a step forward and hard code the priorities in our earlier home alarm system example.
    - Code:

```
while(1){
   if(Smoke_Detected() == TRUE){
      Sound_The_Fire_Alarm();
   }
   else if (Burglar_Detected() == TRUE) Sound_The_Burglar_Alarm();
   }
   else if (switch == PRESSED) { Ring_The_Doorbell();
    }
 }
```

Going Dynamic

- The else clauses will change the schedule to a dynamic one.
  - As long as smoke is detected, the burglar alarm and doorbell will be ignored. Similarly, burglar detection will disable the doorbell.

- This **strict prioritization** may or may not be appropriate for a given system. We may want to ensure some **fairness,** perhaps by limiting how often a task can run.

Task Preemption

- Can one of the tasks interrupt a lower priority task? Or do all tasks run to completion?
- What if a burglar breaks a window a second after the doorbell switch is pressed? We wait 30 seconds till we know there is an intruder?
- No. We can just allow the smoke and burglar detection code to preempt Ring_The_Doorbell. We will need to use a more sophisticated task scheduler which can:
  - (1) preempt and resume tasks.
  - (2) detect events which trigger switching and starting tasks.

Is it Fair?

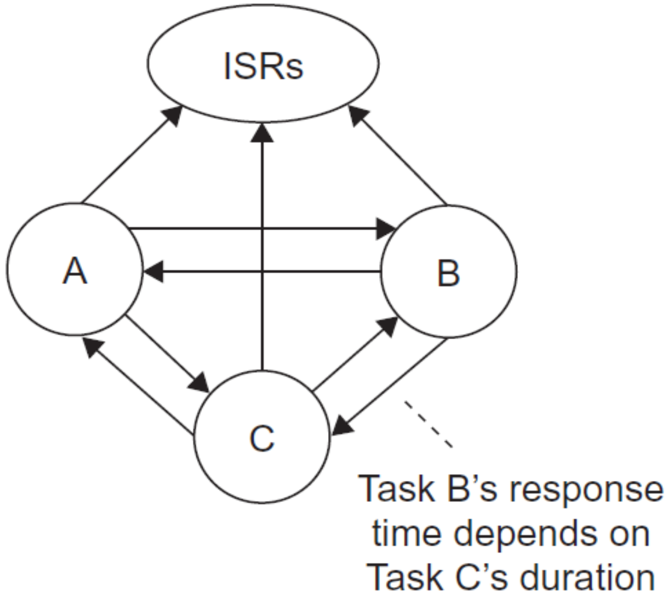
- Prioritizing some tasks over others can lead to starvation of lower priority tasks? They might never get a chance to run.
- This scenario is called as starvation.
- Some ways to feed these lower priority tasks some processor time and keep them happy.

  - We can allow multiple tasks to share the same priority level. If both tasks are ready to run, we alternate between executing each of them (whether by allowing each task to run to completion, or by preempting each periodically).
  - We can limit how often each task can run by defining the task frequency. This is the common approach used for designers of real-time systems. Note that we can still allow only one task per priority level.

Response Time and Preemption

- With the non-preemptive static scheduler, each task's response time depends on the duration of all other tasks and ISRs, so there are nine dependences.

Non-Preemptive
Static Scheduling

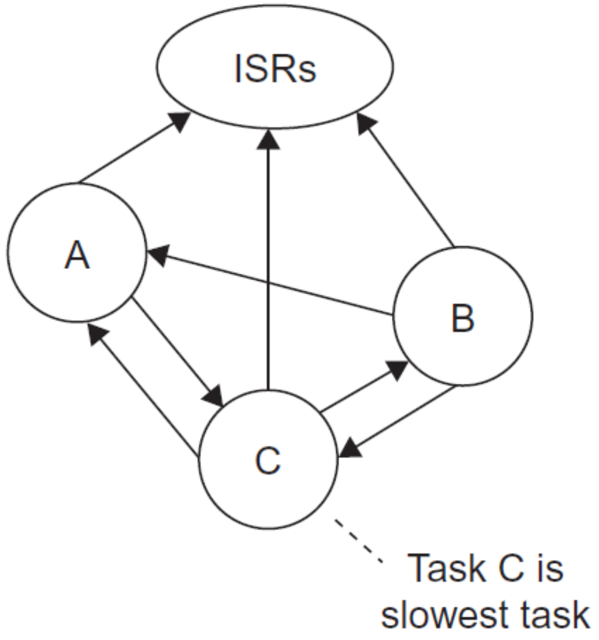


9 dependences
- Higher priority tasks and ISRs
- Lower priority tasks

Response Time and Preemption

- With the non-preemptive dynamic scheduler, we assign priorities to tasks (A B C).
- Any task no longer depends on lower priority tasks, so we have more timing independence and isolation.
- If C starts running, it means a delay for other tasks, A and B each which results in a total of eight dependences.

Non-Preemptive Dynamic Scheduling

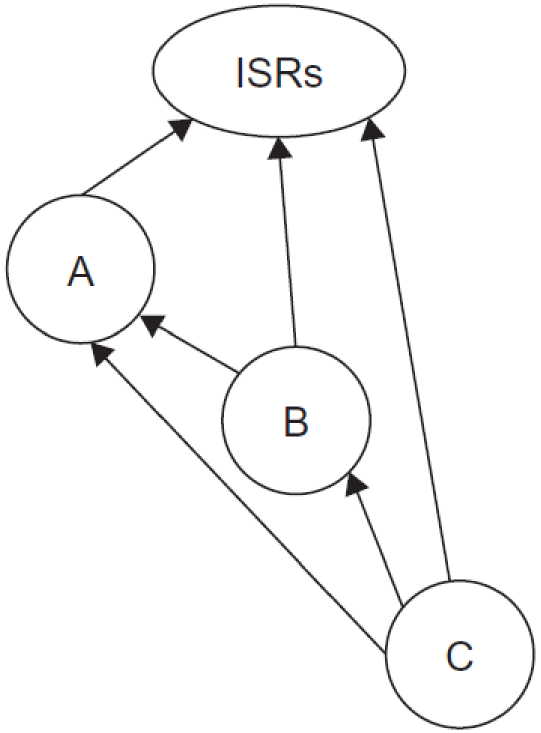


8 dependences
- Higher priority tasks and ISRs
- **Slowest** task

Response Time and Preemption

- With the preemptive dynamic scheduler, we also prioritize the tasks (A B C). Because a task can preempt any lower priority task.
- As a result we have only six dependences.
- This means that in order to determine the response time for a task, we only need to consider **higher priority tasks.**

Preemptive
Dynamic Scheduling



6 dependences
- Only higher priority tasks and ISRs

Stack Memory Requirements

- The non-preemptive approach requires only **one call stack,** while a preemptive approach typically requires **one call stack per task.**

- The function call stack holds a function’s state information, such as return address and limited lifetime variables.

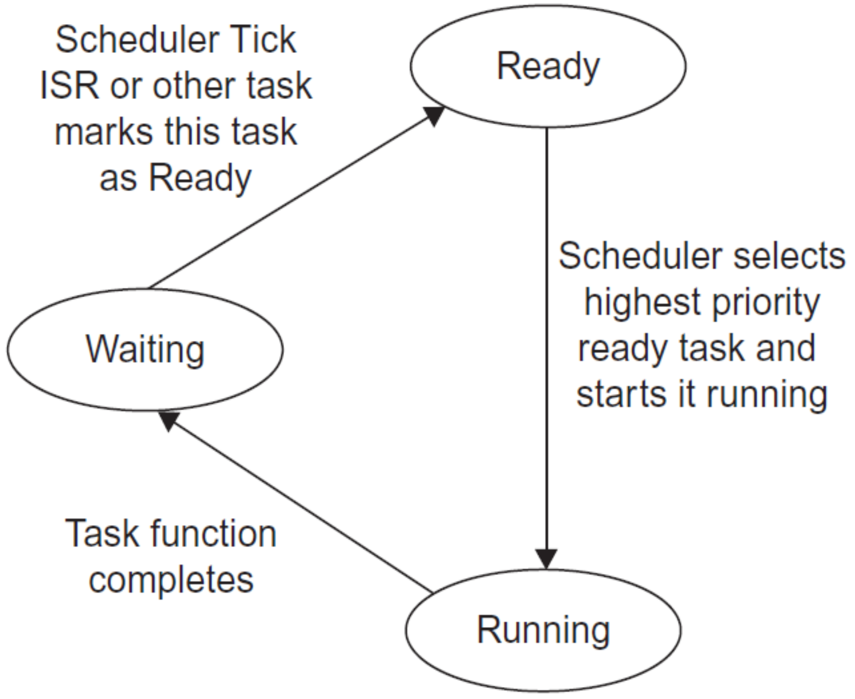
- Without task preemption, task execution does not overlap in time, so all tasks can share the same stack.

Task Management

- A task will be in one of several possible states. The scheduler and the task code itself both affect which state is active.
- With a **dynamic scheduler,** a task can be in any one of the Ready, Waiting, or Running states.
- Task management consists of:
  - Changing states
  - Context switching
  - Sharing Data between tasks

Task States

- **Waiting** for the scheduler to decide that this task is ready to run.
- **Ready to start running** but not running yet. There may be a higher-priority task which is running.
- **Running** on the processor.


Ready
Scheduler Tick ISR or other task marks this task as Ready
Waiting
Scheduler selects highest priority ready task and starts it running
Task function completes
Running

Non-preemptive Dynamic Scheduler

Transitions between States

- The transition from **ready to running:**
  - In a non-preemptive system, when the scheduler is ready to run a task, it selects the highest priority ready task and moves it to the running state.
  - In a preemptive system, when the kernel is ready to run a task, it selects the highest priority ready task and moves it to the running state by restoring its context to the processor.

- The transition from **running to waiting:**
  - In a non-preemptive system, the only way a task can move from running to waiting is if it **completes.**
  - In a preemptive system, the task can yield the processor and request a delay.

Transitions between States

- The transition from **waiting to ready:**
  - In a non-preemptive system the timer tick ISR moves the task by setting the run flag. Alternatively, another task can set the run flag to request for this task to run.
  - In a preemptive system, the kernel is notified that some event has occurred so it moves that particular task from the waiting state to the ready state.

- The transition from **running to ready:**
  - In a non-preemptive system this transition does not exist, as a task cannot be preempted.
  - In a preemptive system, when the kernel determines a higher priority task is ready to run, it will save the context of the currently running task, and move that task to the ready state.

Context Switching for Preemptive Systems

- In preemptive systems, some of these state transitions require the scheduler to save a task's execution context and restore another task's context to ensure programs execute correctly.

- This is called Context Switch.

Context Switching

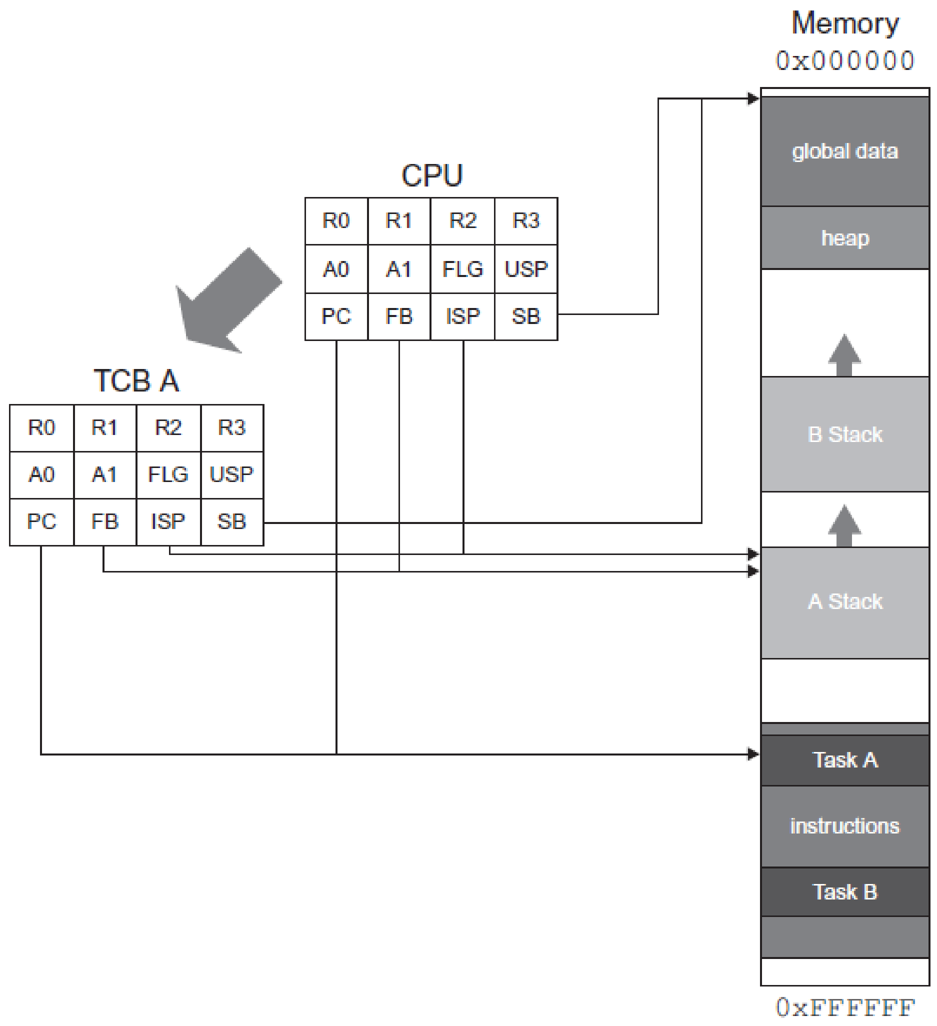
- In order to perform a context switch from one task to another correctly, we must first copy all of this task-specific processor register information to a storage location (TCB of first task).

- Next, we must copy all of the data from TCB of second task into the CPU’s registers.

- Now the CPU will be able to resume execution of the Second task where it left off.

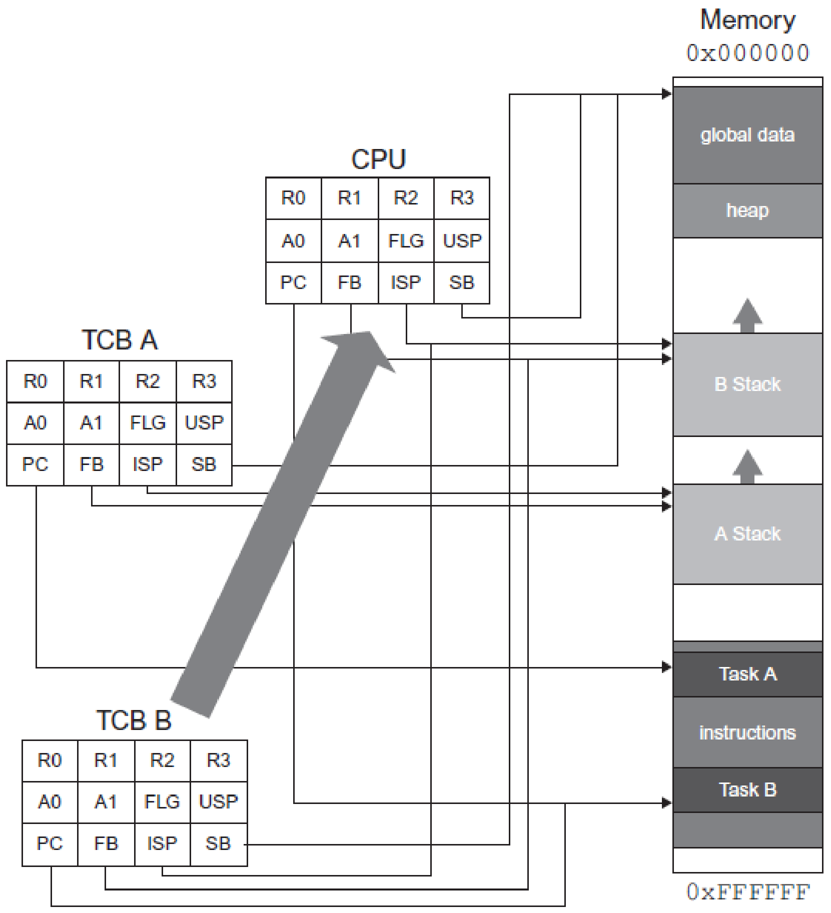
Context Switching

- Saving Task A’s context:



Context Switching

- Restoring Task B's context:



Sharing Data

- Preemption among tasks introduces a vulnerability to data race conditions.
- The system can fail in new ways when:
  - Multiple tasks or ISRs share data, or
  - Multiple instances of a function can execute concurrently
- In order to prevent these failures we need to be careful when considering shared data.

Data Shared Objects

- If a data object is accessed by code which can be interrupted (is not *atomic*), then there is a risk of data corruption.
- *Atomic* code is the smallest part of a program that executes without interruption.

Data Shared Objects

– Code:

```c
unsigned time_minutes, time_seconds;
void task1 (void){
   time_seconds++;
   if(time_seconds >= 60){
      time_minutes++;
      time_seconds = 0;
   }
}

void task2 (void){
    unsigned elapsed_sec;
    elapsed_seconds = time_minutes * 60 + time_seconds;
}
```

Data Shared Objects

- In the code example our shared object is a pair of variables which measure the current time in minutes and seconds.

- If task1 is preempted between lines 4 and 5 or lines 5 and 6, then when task2 runs it will only have a partially updated version of the current time, and elapsed_seconds will be incorrect.

- If task2 is preempted during line 11, then it is possible that time_minutes is read **before** task1 updates it and time_seconds is read **after** task 1 updates it. Again, this leads to a corrupted elapsed_seconds value.

Function Reentrancy

- Another type of shared data problem comes with the use of non-reentrant functions.
  - Code:

```
void task1 ( ){
      . . . . . . . . . . . . . . . .
      swap(&x, &y);
      . . . . . . . . . . . . . . .
}
void task2 ( ){
      . . . . . . . . . . . . . . .
      swap(&p, &q);
      . . . . . . . . . . . . . . .
}
int Temp;
void swap (*i, *j ){
      Temp = *j;
      *j = *i;
      *i = Temp;
}
```

Function Reentrancy

- Such functions are called **non-reentrant.** The code which can have multiple simultaneous, interleaved, or nested invocations which will not interfere with each other is called reentrant code.
- In a multi-processing environment, the non-reentrant functions should be eliminated.

- A function can be checked for its reentrancy based on these three rules:
  - A reentrant function **may not use variables in a non-atomic way** unless they are stored on the stack of the calling task or are the private variables of that task.
  - A reentrant function **may not call other functions** which are not reentrant.
  - A reentrant function **may not use the hardware in a non-atomic way**.

Shared-Data Solutions and Protection

- ***Disable Interrupts***
  - Disable the interrupts during the critical section of the task.
- ***Use a Lock***
- ***RTOS-Provided Semaphore***
- ***RTOS-Provided Messages***
- ***Disable Task Switching***

Disabling Interrupts

- One of the easiest methods is to disable the interrupts during the critical section of the task.
- Once the critical section of the code is executed, the interrupt masking can be restored to its previous state.

```
#define TRUE 1
#define FALSE 0
static int error;
static int error_count;
void error_counter ( ){
     if(error == TRUE){
             SAVE_INT_STATE;
             DISABLE_INTS;
             error_count++;
             error = FALSE;
             RESTORE_INT_STATE;
     }
}
```

Use a Lock

- Associate every shared variable with a lock variable, which is also declared globally.
- If a function uses the shared variable, then it sets the lock variable; and once it has finished process, it resets the lock variable.
- Every function must test the lock variable before accessing it.

```
unsigned int var;
char lock_var;
void task_var ( ){
     unsigned int sum;
      if(lock_var == 0){
              lock_var = 1;
              var = var + sum;
              lock_var = 0;
     }
     else { //message to scheduler to check var
     }
}
```

RTOS-Provided Semaphore

- Most operating systems provide locks to shared variables through the use of *semaphores.*

- A semaphore is a mechanism that uses most of the multitasking kernels to protect shared data and to synchronize two tasks.

Example of Semaphore

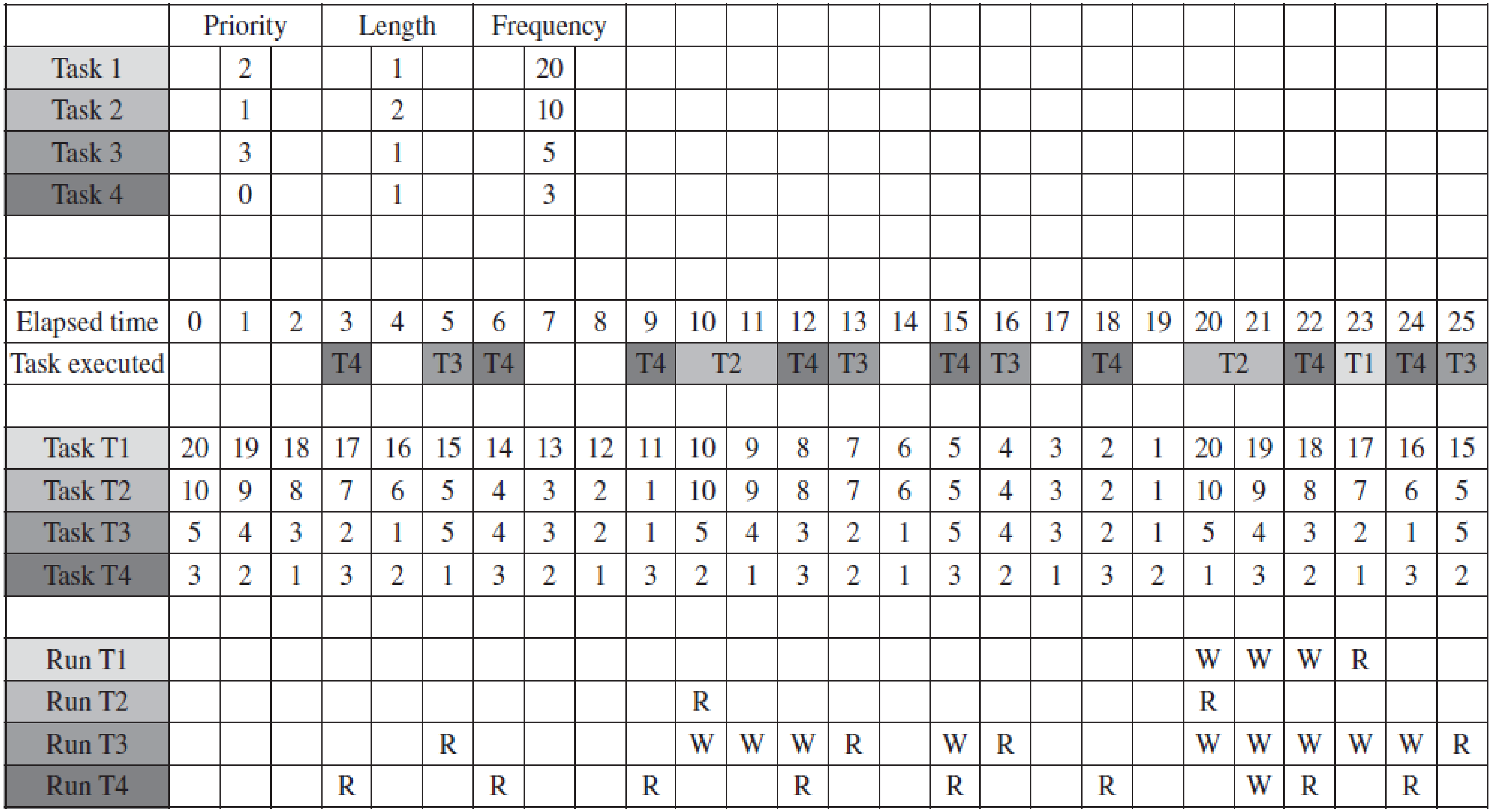
```
typedef int semaphore;
float temp;
semaphore var_temp
void Task1 (void){ wait (var_temp);
    temp = (9/5)(temp + 32); /* Celsius into Fahrenheit */
    Signal (var_temp);
}
void Task2 (void){
    wait(var_temp);
    temp = ADDR0; /* Read ADC value from thermistor */
    temp = ADCtotemp_conversion();
    Signal (var_temp);
}
```

- Just before the tasks enter the critical section, they request the semaphore and only then perform the operation on the shared variable.

Nonpreemptive Dynamic Scheduler

- The scheduler has three fundamental parts.
  - **Task Table:** This table holds information on each task, including:
    - The address of the task's root function.
    - The period with which the task should run (e.g., 10 ticks).
    - The time delay until the next time the task should run (measured in ticks).
    - A flag indicating whether the task is ready to run.
  - **Tick ISR:** Once per time tick (say each 1 millisecond) a hardware timer triggers an interrupt.
  - **Task Dispatcher:** It is simply an infinite loop which examines each task's run flag. If it finds a task with the **run** flag set to 1, the scheduler will clear the **run** flag back to 0, execute the task, and then go back to examining the **run** flags.

Nonpreemptive Dynamic Scheduler



		Priority			Length			Frequency																		
Task 1		2			1			20																		
Task 2		1			2			10																		
Task 3		3			1			5																		
Task 4		0			1			3																		
Elapsed time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Task executed				T4		T3	T4			T4	T2		T4	T3		T4	T3		T4		T2		T4	T1	T4	T3
Task T1	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	20	19	18	17	16	15
Task T2	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5
Task T3	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5
Task T4	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2
Run T1																					W	W	W	R		
Run T2											R										R					
Run T3						R					W	W	W	R		W	R				W	W	W	W	W	R
Run T4				R			R			R			R			R			R			W	R		R	

Example Application Using RTC Scheduler

- Let’s use the RTC scheduler to create a toy with red and green LEDs flashing at various frequencies. The grnLED task toggles a green LED (on board LED 6) every one second, and the redLED task toggles a red LED (on board LED 12) every 0.25 seconds. The grn_redLED task toggles one red LED and one green LED (on board LED 7 and LED 8 respectively) every 0.5 seconds.

Scheduler Code for Adding a New Task

```c
int Add_Task(void (*task)(void), int time, int priority){
        //Check for valid priority
        if(priority >= MAX_TASKS || priority < 0)
                return 0;
        //Check to see if we are overwriting an already scheduled task
        if(GBL_task_table [priority].task != NULL)
                return 0;
        // Schedule the task
        GBL_task_table[priority].task = task;
        GBL_task_table[priority].run = 0;
        GBL_task_table[priority].timer = time;
        GBL_task_table[priority].enabled = 1;
        GBL_task_table[priority].initialTimerValue = time;
        return 1;
}
```

Scheduler Code for Task Timer Initialization

```c
void init_Task_Timers(void){
    int i;
    // Initialize all tasks
    for(i = 0; i < MAX_TASKS; i++){
        GBL_task_table[i].initialTimerValue = 0;
        GBL_task_table[i].run = 0;
        GBL_task_table[i].timer = 0;
        GBL_task_table[i].enabled = 0;
        GBL_task_table[i].task = NULL;
    }
}
```

Scheduler Code Initialization

```
void Init_RTC_Scheduler(void){
     IEN(TMR0,CMIA0) = 0;
     IPR(TMR0,CMIA0) = 3;
     MSTP(TMR0) = 0;
     TMR0.TCNT = 0x00;
     TMR0.TCORA = 78;
     TMR0.TCSR.BYTE = 0xE2;
     IEN(TMR0,CMIA0) = 1;
     TMR0.TCCR.BYTE = 0x0C;
     TMR0.TCR.BYTE = 0x48;
}
```

- A timer is set up to generate an interrupt at regular intervals. Within the interrupt service routine the timer value for each task is decremented.
- When the timer value reaches zero, the task becomes ready to run.

Schedular code for Running Tasks

```c
void Run_RTC_Scheduler(void){
   int i;
   // Loop forever
   while(1){
      // Check each task
      for(i = 0; i < MAX_TASKS; i++){
         // check if valid task
         if(GBL_task_table[i].task != NULL){
            // check if enabled
            if(GBL_task_table[i].enabled == 1){
               // check if ready to run
               if(GBL_task_table[i].run == 1){
                  // Reset the run flag
                  GBL_task_table[i].run = 0;
                  // Run the task
                  GBL_task_table[i].task();
                  // break out of loop to start at entry 0
                  break;
               }
            }
         }
      }
   }
}
```

Code – Tasks and Main

```
void grnLED(void){
    if(LED3 == LED_ON)
        LED3 = LED_OFF;
    else
        LED3 = LED_ON;
}

void redLED(void){
    if(LED4 == LED_ON)
        LED4 = LED_OFF;
    else
        LED4 = LED_ON;
}

void grn_redLED(void){
    if(LED5 == LED_ON)
        LED5 = LED9 = LED_OFF;
    else
        LED5 = LED9 = LED_ON;
}
```

Code – Tasks and Main

```c
void main(void){
   ENABLE_LEDS;
   init_Task_Timers();
   Add_Task(grnLED,10000,0);
   Add_Task(redLED,2500,1);
   Add_Task(grn_redLED,5000,2)
   Init_RTC_Scheduler();
   Run_RTC_Scheduler();
 }
```

In This Chapter We Learned

- Scheduling
- Task Ordering
- Dynamic Scheduling
- Task Preemption and Response time
- Task Management
- Task States
- Transitioning between states
- Context Switching
- Shared Data problems and its solutions
- Example: Non preemptive dynamic scheduler

Renesas Electronics America Inc.