SYS/BIOS On-Line Training

Tasks
SYS/BIOS Multitasking (Task)

- More advanced scheduling technique that uses tasks for scheduling
  - Tasks can block waiting for events
  - Enables another tier for processing events
- Number and state of tasks can dynamically vary over execution timeline
- Task priorities can be changed dynamically
- Scheduling is achieved by programmatically calling the multitasking kernel
  - Priority levels
  - State of readiness
Task

- A logically complete program segment that includes:
  - Pointer to a function
  - Argument list
  - Priority
    - Can be changed during runtime
  - Pointer to a unique task stack
    - Stores local variables
    - Nested function calls

- Interrupts run on the system stack

Each task has its own stack! This allows for the blocked state
Tasks Differ From Swis

Swi

Swi cannot pend.  
Swi always returns from function.

Task

Task only returns when no longer needed.  
Otherwise normally an infinite loop.
Task Lifecycle
Tasks are Ready to Run When Created
Tasks are Preemptive

SYS/BIOS Startup

READY

RUNNING

Preemption
Tasks Can Be Blocked

SYS/BIOS Startup

READY

PREEMPTION

RUNNING

Task is readied

Task is blocked

BLOCKED

Semaphore_pend

Semaphore_post
Tasks Terminate Automatically

SYS/BIOS Startup

READY

RUNNING

BLOCKED

TERMINATED

Preemption

Task is readied

Task exits

Task is blocked

Return from task’s function

Return from task
Tasks Can Be Deleted Prior to Completion

SYS/BIOS Startup

READY

RUNNING

BLOCKED

TERMINATED

Task is deleted

Preemption

Task is readied

Task exits

Task is blocked

Task is deleted

*
Task Run-Time API

```c
Task_Params taskParams;
Task_Handle myTask;
Error_block eb;

Task_Params_init(&taskParams);
  taskParams.stackSize = 512;
  taskParams.priority = 15;
Error_init(&eb);
myTask = Task_create((Task_FuncPtr)hiPriTask, &taskParams, &eb);
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task_create</td>
<td>Allocate and initialize a new instance object and return its handle</td>
</tr>
<tr>
<td>Task_delete</td>
<td>Finalize and free this previously allocated instance object, setting the referenced</td>
</tr>
<tr>
<td>Task_disable</td>
<td>Disable the task scheduler</td>
</tr>
<tr>
<td>Task_exit</td>
<td>Terminate execution of the current task</td>
</tr>
<tr>
<td>Task_getEnv</td>
<td>Get task environment pointer</td>
</tr>
<tr>
<td>Task_getIdleTask</td>
<td>Returns a handle to idle task object</td>
</tr>
<tr>
<td>Task_getMode</td>
<td>Retrieve the Mode of a task</td>
</tr>
<tr>
<td>Task_getPri</td>
<td>Get task priority</td>
</tr>
<tr>
<td>Task_Params_init</td>
<td>Initialize this config-params structure with supplier-specified defaults before</td>
</tr>
<tr>
<td>Task_restore</td>
<td>Restore Task scheduling state</td>
</tr>
<tr>
<td>Task_self</td>
<td>Returns a handle to the currently executing Task object</td>
</tr>
<tr>
<td>Task_setEnv</td>
<td>Set task environment</td>
</tr>
<tr>
<td>Task_setPri</td>
<td>Set a task's priority</td>
</tr>
<tr>
<td>Task_sleep</td>
<td>Delay execution of the current task</td>
</tr>
<tr>
<td>Task_stat</td>
<td>Retrieve the status of a task</td>
</tr>
<tr>
<td>Task_yield</td>
<td>Yield processor to equal priority task</td>
</tr>
</tbody>
</table>
Creating a Task With XGCONF

### module Task

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>numPriorities</td>
<td>16</td>
<td>Number of Task priorities supported. Default is 16</td>
</tr>
<tr>
<td>defaultStackSize</td>
<td>512</td>
<td>Default stack size (in MAUs) used for all tasks</td>
</tr>
<tr>
<td>defaultStackSection</td>
<td>.taskStackSection</td>
<td>Default memory section used for all statically created task stacks</td>
</tr>
<tr>
<td>defaultStackHeap</td>
<td>null</td>
<td>Default Mem heap used for all dynamically created task stacks</td>
</tr>
<tr>
<td>enableIdleTask</td>
<td>true</td>
<td>Create a task (of priority 0) to run the Idle functions in</td>
</tr>
<tr>
<td>idleTaskStackSize</td>
<td>512</td>
<td>Idle task stack size in MAUs</td>
</tr>
<tr>
<td>idleTaskStackSection</td>
<td>.taskStackSection</td>
<td>Idle task stack section</td>
</tr>
<tr>
<td>idleTaskVitalTaskFlag</td>
<td>true</td>
<td>Idle task’s vitalTaskFlag. (see @link #vitalTaskFlag)</td>
</tr>
<tr>
<td>allBlockedFunc</td>
<td>ti.sysbios.kni.Task/allBlocked...</td>
<td>Function to call while all tasks are blocked</td>
</tr>
<tr>
<td>initStackFlag</td>
<td>true</td>
<td>Initialize stack with known value for stack checking at runtime (see @link #checkStackFlag)</td>
</tr>
<tr>
<td>checkStackFlag</td>
<td>true</td>
<td>Check ‘from’ and ‘to’ task stacks before task context switch</td>
</tr>
</tbody>
</table>

### Fxn: Task Function

- **argx**: values passed to task function
- **stack**: pointer to task stack
- **stackSize**: default = 512
- **priority**: 1-15, and -1 means task stays inactive
- **stackheap**: null means that default stack heap is used
Task Stack Overflow

• What size of a stack is needed for a task?

• If Stack size too small
  – May overflow if too many nested calls
  – Or is too many local variables

• If Stack size too large
  – Wastes memory

• Monitor task stack usage on exit to check for wasted space
  – Use ROV or Task_stat()
  – Returns the maximum amount of stack used by a task

• Detect overflow by setting Task.checkStackFlag=TRUE
  – By default, the Task module checks to see whether a Task stack has overflowed at each Task switch