DSP II: ELEC 4523

TSK and SEM Modules

Objectives

- Become familiar with TSK and SEM modules and their use

Reading

- SPRU423 TMS320 DSP/BIOS Users Guide: Tasks (section), Semaphores (section)
- PowerPoint Slides from DSP/BIOS II Workshop: Module 7
- Code Composer Studio Online Help: TSK Module, SEM Module

Lab Module Prerequisites

None.

Description

TSK Module

Tasks are independent threads of code that conceptually run concurrently. The processor time is shared among the tasks. Each task has a priority which is used to determine which task gets processor time.

Each task has its own stack to store local variables, nesting function calls and for saving the task state when it is preempted. The stack size can be set individually for each task.

A task is always in one of four states

- **Running**, which means the task is the one actually executing on the system's processor;
- **Ready,** which means the task is scheduled for execution subject to processor availability;
- **Blocked,** which means the task cannot execute until a particular event occurs within the system; or
- **Terminated,** which means the task is "terminated" and does not execute again.

Figure 1 shows a state transition diagram for a task. There is only one thread in the TSK_RUNNING state. When a task is in the TSK_RUNNING state all the other tasks in the TSK_READY state are at the same or lower priority. When a task of higher priority enters the TSK_READY state a preemption immediately occurs and the running task enters the TSK_READY state and the higher priority task enters the TSK_RUNNING state. Tasks can become blocked when a resource is unavailable or when some other event that causes blocking occurs. Semaphores are used to synchronize access to resources and can cause a task to block. When the resource becomes available the task enters the TSK_READY state.

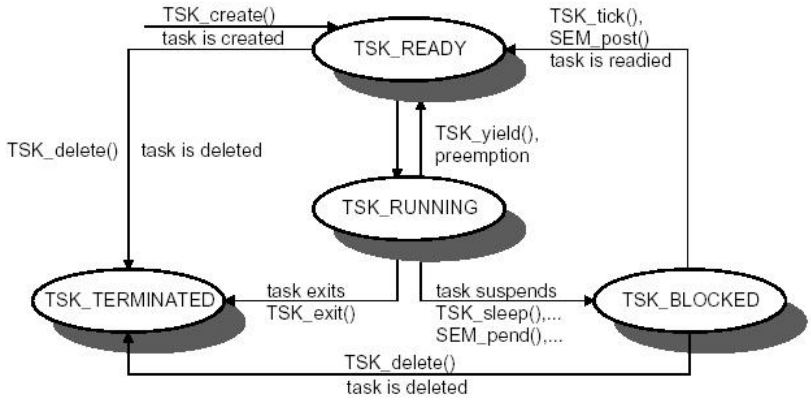

Figure 1: State transition diagram for a task, taken from *SPRU423 TMS320 DSP/BIOS Users Guide*

To create an TSK open the configuration file, right click on Scheduling->TSK and select Insert TSK. This will add a new TSK object. You can right click on the object and select Properties to change its properties. Set the TSK priority on the General tab. Click on the Function tab and put the function name in for the function you want to handle the TSK. Figure 2 shows a view of the configuration file where the TSKs can be seen in their corresponding priority. Notice that the `TSK_idle` is priority 0 and it is reserved. This is the idle task. When your program runs the tasks with the same priority will get initialized in the order that they are shown in the configuration tool.

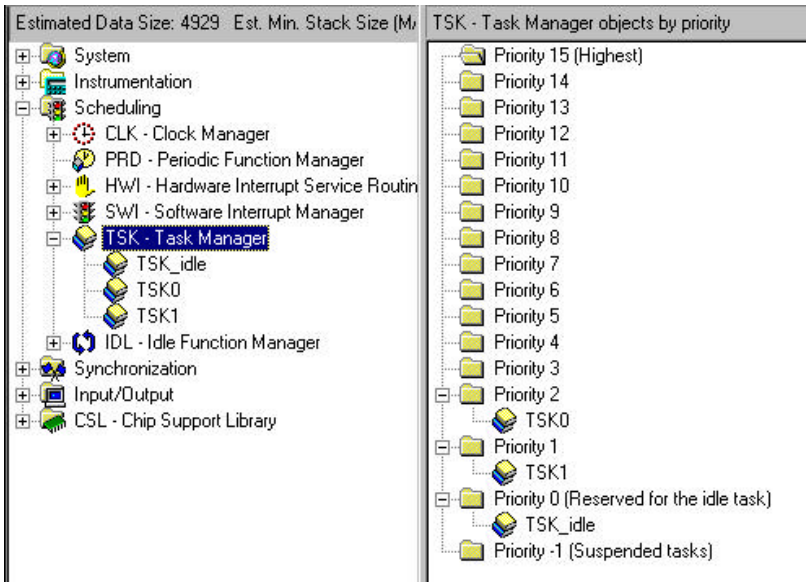

Figure 2: Configuration file showing TSKs and priorities

There are many functions in the TSK module. Some of the more commonly used ones are:

- `TSK_disable` - Disable DSP/BIOS task scheduler
- `TSK_enable` - Enable DSP/BIOS task scheduler
- `TSK_yield` - Yield processor to equal priority task

SEM Module

Semaphores provide a way for intertask communication and synchronization. They are used to coordinate access to a shared resource being accessed by multiple tasks. The SEM objects are counting semaphores which means they keep a record of the number of corresponding resources available.

There are two main functions for using a semaphore once it has been created.

- `SEM_pend` is used to wait for a semaphore. The timeout parameter to `SEM_pend` allows the task to wait until a timeout, wait indefinitely (`SYS_FOREVER`), or not wait at all. `SEM_pend`'s return value is used to indicate if the semaphore was signaled successfully.
- `SEM_post` is used to signal a semaphore. If a task is waiting for the semaphore (in the TSK_BLOCKED state), `SEM_post` removes the task from the semaphore queue and puts it on the ready queue. If no tasks are waiting, `SEM_post` simply increments the semaphore count and returns.

To create an SEM open the configuration file, right click on Scheduling-> SEM and select Insert SEM. This will add a new SEM object. You can right click on the object and select Properties to change its properties.

Laboratory

Part 1

- Create a new project called `tsklab`.
- Create a new DSP/BIOS Configuration file and use the `C6xxx.cdb` template for use with the simulator.
- Save the file as `tsklab.cdb` and add it in your project. Also add the `tsklabcfg.cmd` file.
- If using the simulator then change the RTDX interface to Simulator by right clicking on Input/Output->RTDX and bringing up the properties. Change the RTDX mode to Simulator. If you do not do this then when you load your program you will see the error "RTDX application does not match emulation protocol." If you are loading onto an EVM or DSK you shouldn't need to change this.
- Create a LOG object by right clicking on Instrumentation->LOG and selecting Insert LOG. Change the name to `trace`. Set its properties to have a length of 128 and be a fixed buffer.
- Create two TSKs with the following properties
  - Name: `TSK0`, priority: 1, function: `_funTSK0`.
  - Name: `TSK1`, priority: 1, function: `_funTSK1`.
- Create a main.c file and include a `main` function that does nothing. Include this file in your project.
- In the main.c file make functions for your TSKs, `funTSK0` and `funTSK1`. Each function should print to the `trace` LOG object once at the beginning of the function to indicate that the particular TSK is starting and once at the end of the function to indicate that the particular TSK is ending.
- Add code to each task that loops 5 times and prints to trace the task number and the loop number each time through the loop.
- The basic structure of main.c should be:

```
#include <std.h>
#include <tsk.h>
#include <log.h>
#include "tsklabcfg.h"

void main()
{
}

void funTSK0()
{
      /* code here */
}

void funTSK1()
{
      /* code here */
}
```

- Build and load your project.
- Open the log view, DSP/BIOS->Message Log.
- Run the program and record the results.

Part 2

- Copy the `main.c` file from above to a new file `main2.c`. Remove the `main.c` from your project and add `main2.c`.
- Change the code so that the two tasks yield to each other within the loop after the print statement.
- Build and load your project.
- Run the program and record the results.
- How does the execution change?

Part 3

- Set one of the tasks to a higher priority than the other.
- Build and load your project.
- Run the program and record the results.
- Does the execution appear different from part 2?

Part 4

- In this part you will add a semaphore to the program.
- Copy the `main2.c` file from above to a new file `main4.c`. Remove the `main2.c` from your project and add `main4.c`.
- Create a semaphore, `SEM0`, in the configuration file and initialize it to a value of 1.
- Put the tasks at the same priority level 1.
- Remove the task yield from each task.
- Have one task pend on the semaphore and the other post the semaphore. Use `SYS_FOREVER` for the timeout on the pend.

- Put the task that pends on the semaphore first in the priority list in the configuration file.
- Build and load your project.
- Run the program and record the results.

Part 5

- Change the semaphore initial value to 2, 3, etc.
- Build and load your project.
- Run the program and record the results.
- What happens with the different initial values?

Part 6

- Make the task that is posting a higher priority than the other.
- Build and load your project.
- Run the program and record the results.

Part 7

- Reverse the priority on the tasks.
- Build and load your project.
- Run the program and record the results.