CS 4390/5372: Specification and Design of Real-Time Systems
Lab1: Hardware and Execution Control

In this experiment, you will:

- Use the ARM Board or vxSim as assigned in Lecture 3.
- Use the Workbench to create project in host-target environment
- Specify task parameters such as priority and stack size,
- Get task parameter information about tasks running on the system,
- Identify when a stack crash has occurred,
- Control whether a task is suspended, delayed or running.

Requirements:

We shall use Windriver's Workbench and the hardware target. However, you may also experiment with using simulated target environment using vxsim.

Use one-side paper and single spacing for the report.

1. Complete Part A and B experiments. Record all pertinent commands that you have executed and their results. Attempt to understand and explain the significance of each step.
2. Prepare lab report using the prescribed format. Include what you performed at each phase of the lab and what results you received. In the report identify by each step of the experiment letter and number (e.g. A7, B3) and respond to all underlined questions. Include descriptions of all procedures/activities, results, and observations, specifically the shell commands and their outcomes.

PART A: TASK STACK

A1. Use files *recur.c* and *recurTwo.c* (with three executable functions: *recur*, *recur1*, and *recur2*). Create downloadable project, compile and download the files. **How do you accomplish it?**

```
/* recur.c */
#include "vxWorks.h"
int count;
void recur(int level)
{
        if (level>0)
        {
                count++; recur(level-
                1);
        }
}
```

```
/* recurTwo.c */
#include   "vxWorks.h"
int count1;
int count2;
void recur1(int level)
{
            if (level>0)
            {
            count1++;
            recur1(level-1);
             }
}
void recur2(int level)
{
            if (level > 0)
            {
            count2++;
            recur2(level-1);
            }
}
```

A2. Open and view the file *recur.c* in editor. Load *recur.o.* **Explain two ways of loading an object file to the target (hint: lookup the 'ld' command).**

A3. Function **recur()** increments **count** every time it is called. Call **recur()** with a small value of its argument (**level** equals to e.g. 4, 5, or 6) and check the value of **count**.. **How can you do it? How would you set a variable defined in the program to a predefined value?**

A4. Using function **repeat** create a task that loops forever, calling **recur** with a **level** = 6. Use shell syntax: **-> tid = repeat(0,recur,6).** The return value of **repeat()** (i.e. **tid** – the task ID of the task created) will be used later. Check that the **count**. **How do you do it? Comment of the received result.** NOTE: To re-execute any shell command, you could use the command recall: **[Esc]**, **k** and then **[Enter]**.

A5. Examine the task using **ti(tid) –** where **tid** is the task ID from point A4. **What is the task name, entry point, identifier, priority, and stack size?**

A6. Using the **i()** command three or four times, watch the stack pointer change. **Note and explain.**

A7. Trace the stack frames using shell command **tt** to watch the progress of the program. Note that the task is suspended while **tt()** examines its stack. **Explain what you observe.**

A8. Examine the stack usage using **checkStack(tid)**. **Explain the values you received (total size, used size, largest used, margin)**.

A9. Delete the task started in point 5. **What shell command would you use? How do you know you were successful in deleting the task?**

A10. Create another task with small stack of size 1000 bytes to execute **recur()** with a larger argument. This will cause a stack overrun or stack crash. Use: -> tid=taskSpawn("tBadTask",75,0,1000,repeatHost,0,recur,100) If you get a prompt back (you may not as the system may crash, see note below) check the stack. -> checkStack(tid)

Note:

The margin of zero (and the invalid or missing task name) indicates a stack overrun. Since we do not know what was stored in the memory below the stack we do not know what was corrupted. It is possible that the system appears to work fine for a while and then fails, when the corrupted memory is finally used. ***Reboot the target before continuing!***

On VxSim, you may not appear to have overrun the stack, i.e. the margin may still be nonzero. If, however, the high water mark exceeds the amount of stack space requested initially (1000 bytes in the ***taskSpawn()*** *call as in the point above), you should consider a stack overrun to have occurred. The reason is that the extra 8000 bytes that VxSim adds to task stacks are needed for simulated interrupt handling. Even if the interrupt handling mechanism has not already written beyond the end of the task's (inflated) stack1, it is likely to do so in the future.*

PART B: TASK CONTROL AND DELAYS

B1. Load *recurTwo.o*. Create a task **tid1** that loops forever calling **recur1()** (defined within the recur2.o module). Syntax: **repeat(***number_of_repeats, function, argument***)**. **Specify the actual command executed and verify that the task is running by examining the variable count1 at least three times**.
B2. Suspend the task using created above and verify that it is not running by observing value of **count**. **Subsequently, resume the task. Identify the shell commands you must use to suspend the task, observe the value of variable, and resume the task.**
B3. Using **repeat** create another task **tid2** calling **recur2()** that loops forever. Observe the tasks status using **->i()**. Note the status and the priority of both tasks. Check the values of both **count1** and **count2**. **Note that count1 is incremented but not count2. Why?**
B4. Lower the priority of **tid1** and check the values of **count1** and **count2**. **What shell command and argument you need to use? Note that now tid2 is running (count2 is incremented) while tid1 is not (count1 is not incremented).Why?**
B5. Suspend **tid2** and re-check the values of **count1** and **count2**. **Note that tid1 is now running, but not tid2. Why?**
B6. Resume **tid2** and restore **tid1** to the same priority 100. **What shell command and arguments are you using?**
B7. To have both tasks share the CPU, enable round-robin scheduling with time slice 10 ticks using **-> kernelTimeSlice(10). Verify that both count1 and count2 are being incremented. How do you do it?**
B8. **Clean up deleting both tasks and setting back priority scheduling. What instructions do you use?**
B9. You can "delay the shell" using **-> taskDelay(600)**. **Explain what happens. What if you had inadvertently typed -> taskDelay(6000)? What options (other than rebooting the target) do you have to get the shell back?**
B10. Create a task implementing delay by executing **-> tid=sp(taskDelay,100000**). **What specific shell command do you use to get information about this task? Re-execute the command couple of times and comment on the value the delay counter.**

Reboot Hardware Target !!!