Experiment #2
Multi-Tasking

Introduction

Modern real-time systems are based on the complementary concepts of multitasking and intertask communications. A multitasking environment allows real-time applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The intertask communication facilities allow these tasks to synchronize and coordinate their activities.
The VxWorks multitasking kernel, *wind*, uses interrupt-driven, priority- based task scheduling. It features fast context switch time and low interrupt latency.

Objectives

The following are the primary objectives of this experiment:

- To teach the student how to initiate multiple processes using Vxworks tasking routines.

Description

Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on a basis of ascheduling algorithm. Each apparently independent program is called a *task*. Each task has its own *context*, which is the CPU environment and system resources that the task sees each time it is scheduled to run by the kernel.
On a *context switch*, a task's context is saved in the *Task Control Block*(TCB). A task's

context includes:

- a thread of execution, that is, the task's program counter
- the CPU registers and floating-point registers if necessary
- a stack of dynamic variables and return addresses of function calls
- I/O assignments for standard input, output, error
- a delay timer
- a timeslice timer
- kernel control structures
- signal handlers
- debugging and performance monitoring values

1. Task Creation and Activation

The routine *taskSpawn* creates the new task context, which includes allocating and setting up the task environment to call the main routine(an ordinary subroutine) with the specified arguments. The new task begins at the entry to the specified routine.

The arguments to *taskSpawn()* are the new task's name(an ASCII string), priority, an "options" word(also hex value), stack size(int), main routine address(also main routine name), and 10 integer arguments to be passed to the main routine as startup parameters.

2. Syntax

id = taskSpawn(*name,priority,options,stacksize,function, arg1,..,arg10);*

3. Example

This example creates ten tasks which all print their task Id once:

```
------------------------------------------------------------------------
#define ITERATIONS 10

void print(void);

spawn_ten() /* Subroutine to perform the spawning */
{
int i, taskId;
for(i=0; i < ITERATIONS; i++)  /* Creates ten tasks */
          taskId = taskSpawn("tprint",90,0x100,2000,print,0,0,0,0,0,0,0,0,0,0);
}

void print(void)    /* Subroutine to be spawned */
{
printf("Hello, I am task %d\n",taskIdSelf()); /* Print task Id */
}
------------------------------------------------------------------------
```

Procedures

1. Copy the source code in the example and compile it.
2. Load the object file onto the target machine.
3. Run the example by executing "spawn_ten" on the WindSh.

Note: Make sure you have redirected I/O, otherwise you won't see the results of the printf commands.

Follow On Experiment

As shown in the example, the task creation process allows the passing of ten arguments to the *function*.

Experiment 1. As discussed above, up to a ten integer arguments can be passed to a task during the call to *taskSpawn()*. Pass a unique argument to each of the ten tasks and have them print it. For example, pass the increment variable "i" in subroutine spawn_ten().

Experiment 2. Assign each task a unique priority. Is there any difference in output? Has the order in which the ten tasks print changed? If not, explain why.

Additional Information

Refer to the VxWorks Programmer's Manual and Reference Manual.
