CS 4390/5372: Real-Time Systems
Lab4: Watchdog and Function Timing
Due: Friday 8/4/2017
(By 11:59 pm. E-mail to yyjacquez@gmail.com)

Introduction:

In a real-time system, the correctness of the system depends on having the timely arrival of results or an expected action executed. It is very important that we know how to determine the execution time of a segment of application code to give us quantitative information for performance benchmarking and scheduling analysis. Also you must know the capabilities and limitations of different time measurement routines available on a real-time system.

In this experiment, you will:

- To learn the concept of watchdog timer
- Use POSIX 1003.1b and VxWorks OS routines to delay task execution

A popular timing facility is a watchdog timer. On a typical hardware target, there are usually one or more hardware timers that interrupt the CPU periodically. At each timer interrupt, VxWorks executes the associated interrupt service sub-routines (ISR) to perform bookkeeping jobs. The system clock, with an ISR responsible for incrementing the tick count (16.67 milliseconds per tick increment), updates delays and timeouts (*taskDelay* and timeout values specified in *semTake*, *msgQReceive*, etc.), and checks for time slice rescheduling (*kernelTimeSlice*).

VxWorks provides the routine wdStart() to register your C function as part of the system clock ISR. The code example below illustrates its usage.

wdStart(tPtr->wdVSense, CAR_PRESENT_WAIT_TIME*sysClkRateGet(), (FUNCPTR)goYellow, light_ptr);

The argument to this routine are watchdog ID, time delay in ticks, address of the C function, and an argument to be passed to the C function. There are two things to watch out for when calling *wdStart()*:

- A watchdog ID can only be associated with one C function and time delay at any one time. So if the timer associated with *wdVSense* (see above) has not expired yet, and you called *wdStart* again with the same ID, the on-going timer would be cancelled and replaced with the latest arguments passed by *wdStart*. For every independent watchdog timer you need, simply create a new watchdog ID for it.
- The attached C function has to have one integer parameter and VOID as a return type. Also, certain restrictions are placed on what routines you may call within an ISR. More on this in the hardware interrupt section.

When a watchdog timer expires, the associated function is executed immediately as interrupt service code at the interrupt level of the system clock (or is placed on the *tExcTask* work queue if the kernel is not able to execute it immediately for any reason).

Watchdog functions are provided by the wdLib watchdog library.

- **wdCreate()** – create a watchdog timer
- **wdDelete()** – delete a watchdog timer
- **wdStart()** – start a watchdog timer
- **wdCancel()** – cancel a currently counting watchdog

A watchdog timer must be created - *wdCreate()* and started - *wdStart()*. The watchdog can be canceled before it expires - *wdCancel()*. Check the *wdLib* manual page for detailed syntax.

The included file *wdog.c* include routine *wd_test(wdID)*, which logs an appropriate message and restarts the watchdog timer to invoke the routine again after some random time period. The watchdog executes on the system interrupt level, and not as a task on the application priority level. It is necessary to create the *wdID* watchdog timer before the routine is executed. We use a simple function *random(int)* to generate uniformly distributed number - the standard C *rand* function returns an integer value between *0* and *RAND_MAX.*

VxWorks functions *timex* and *timexN* may be used (also from the shell line) to determine execution time of a function (or a group of functions).

Requirements:

We shall use Workbench (IDE) and the hardware target. However, you may also experiment with using simulated target.

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, the shell commands and their outcomes.
3) Attach to your report only any modified source code (highlight/comment the modified sections).

Part A:

A1. Add the example source code to the project, compile, and download to the target. Start the shell.

A2. Execute function *wd_test(wdID)* from command line. Observe the output and the currently running tasks (*i*). **Where is the output displayed? What tasks are running? Explain.** Create required watchdog timer *wdID = wdCreate()* and reexecute. Check the status of watchdog using "*show wdID*". **Show the results and comment on them.**

A3. Experiment with *wdCancel*, *wdStart*, and *wdDelete* from the shell line. **Explain and show your results.**

A4. What VxWorks system function you must use to display clock resolution? **Use it, show result and explain.**

A5. Describe the experiments showing the VxWorks shell command lines required to: (a) add *val = 23.12* to the symbol table, (b) create buffer of 30 characters and fill it with string "*value is XX.XX*" (where XX.XX is the numerical value of variable *val*), (c) start the watchdog to print the string with 5 seconds delay. **Show your commands and results.**

Part B:

B1. Add program timing.c to the project. **Explain the functionality of functions: *ifac(), function_to_time(), p_timing()*, and *v_timing()*.**

B2. Execute *function_to_time()* in WindSh with different arguments. For example, type *function_to_time(10)* to specify a value of 10 for the integer argument *howmany*. **What shell command you must use? What are the results?**

B3. Execute *v_timing()* and *p_timing()* functions with an argument of 200 five times. DO NOT INCLUDE RAW OUTPUT IN THE REPORT. **Are the results the same? Where do you get the information on the timing? What are the units? What is the average?**

B4. Repeat experiment B2 for different arguments (use e.g. 100 and 300) recording the execution time. Use both *p_timing()* and *v_timing()*. **Build a table repeating each experiment ten times and computing the average and standard deviation for each of the timing method.**

B5. What are the clock resolutions observed using the POSIX and VxWorks routines? Confirm your answers based on the above experiments.

B5. Repeat experiment B2 with smaller arguments (try 20, 10, 5, 2) for both *p_timing()* and *v_timing()*. **What happens? What do you need to modify in the *v_timing()* function to get the timing in the case when the message says: "...execution time too short to be measured meaningfully..."?**

Appendix:

A) wdog.c

```
/* wdog.c */
#include <vxWorks.h> /* Always include this as the first thing in every program */
#include <stdio.h> /* we use printf */
#include <sysLib.h> /* we use sysClk... */
#include <taskLib.h> /* we use tasks... */
#include <wdLib.h> /* we use watchdog */
#include <logLib.h> /* we use logMsg */
#include <stdlib.h> /* we use rand */

#define PERIOD1 (sysClkRateGet()) /* one second period */

WDOG_ID wdID; /* Id for the watchdog used - create it from shell */
int rand_period;

/* function prototypes */
int random(int);
```

```
/* subroutine re-executed when watchdog expires */
/* uses randomly generated period for restart */
/* we shall print the task name instead of taskId */
void wd_test(WDOG_ID wdID)
{
  rand_period = random(PERIOD1);
  printf("WD task %s; restart after %d ticks;\n",taskName(taskIdSelf()), rand_period);
/* we shall restart the watchdog after rand_period time ticks */
  wdStart (wdID, rand_period,(FUNCPTR) wd_test,(int) wdID);
}
```

```
/* a primitive random function returning an integer 0 - arg */
int random(int arg)
{ return ((int)(arg*((float)rand()/(float)RAND_MAX)));
}
```

timing.c

```
/* timing.c */
#include <vxWorks.h> /* Always include this as the first thing in every program */
#include <stdio.h> /* we use printf */
#include <time.h> /* we use clock_gettime */
#include <timexLib.h> /* we use timex */
/*function prototypes*/
void function_to_time(int howmany);
#define NANOS_IN_SEC 1000000000
#define NANOS_PER_MICRO 1000
#define NANOS_PER_MILLI 1000000
```

```
void v_timing(int howmany) /* Function to perform the timing using VxWorks */
{
timex((FUNCPTR)function_to_time,howmany,0,0,0,0,0,0,0);
}
```

```
long ifac(long i)
{
    if (i == 0 || i == 1) return 1;
          else return i*ifac(i-1);
}
```

```
void function_to_time(int howmany) /*just a time wasting – but what does it compute??? */
{
  int i,j,k;
  double sum=0.0;
  for (i=0;i<howmany;i++)
    for (k=0;k<500;k++)
     {sum = 1.0;
       for (j=1;j<10;j++)
          sum = sum + 1.0/(double)ifac(j);}
   printf("The magic number is %7.5f\n",sum);
}
```

```
int p_timing(int howmany) /* Function to perform the timing using POSIX */
{
   struct timespec tpstart;
   struct timespec tpend;
   long long int timedif;

  /* get time before and after */
  clock_gettime(CLOCK_REALTIME, &tpstart);
  function_to_time(howmany); /* timed code goes here */
  clock_gettime(CLOCK_REALTIME, &tpend);

  /* time difference in nanosec */
   timedif = NANOS_IN_SEC*(tpend.tv_sec - tpstart.tv_sec) + tpend.tv_nsec - tpstart.tv_nsec;

  /* print the difference in microseconds and milliseconds */
  fprintf(stderr, "Execution Time: %10.3f microseconds / %8.3f milliseconds\n",
  ((float)timedif/(float)NANOS_PER_MICRO), ((float)timedif/(float)NANOS_PER_MILLI));

  return(0);
}
```