**CS 4390/5372: Real-Time Systems**

**Lab3: Communication and Message Queues**

**Due: Friday 07/21/2017**

**(By 5:59 pm. E-mail to yyjacquez@gmail.com)**

**Introduction:**

We have learned how to use semaphores to preserve mutually exclusive access to shared data. Semaphores do not convey the data but only serve as synchronization tool. Other operating system constructs are used for communication. In VxWorks, two main mechanisms for the intertask communication, designed to convey data between tasks, are pipes and message queues. Message queues allow a variable number of messages, each of variable length, to be queued (FIFO or priority based). Any task can send a message to a message queue, and any task can receive a message from a message queue. Multiple tasks can send to and receive from the same message queue. Two way communications between two tasks generally requires two message queues, one for each direction.

**Objectives:**

The following are the primary objectives of this experiment:

* To introduce time stamp
* To teach how multiple tasks can send/receive data.
* To demonstrate the use of VxWorks message queues and pipes.

**Description:**

The VxWorks *mesgQLib* library provides messages that are queued in FIFO order, with a single exception: there are two priority levels, and messages marked as high priority (urgent) are attached to the head of the queue.

A message queue is created with *msgQCreate()*. Its parameters specify the maximum number of messages that can be queued in the message queue, the maximum length in bytes of each message, and the queue options.

A task sends a message to a message queue with *msgQSend()*. If no tasks are waiting for messages on that queue, the message is added to the queue's buffer of messages. If any tasks are waiting for a message from that message queue, the message is immediately delivered to the first waiting task.

A task receives a message from a message queue with *msgQReceive()*. If messages are already available in the queue's buffer, the first message is immediately dequeued and returned to the caller. If no messages are available, then the calling task blocks and it is added to a queue of tasks waiting for messages. The queue of waiting tasks can be ordered either by task priority or FIFO, as specified when the queue is created.

**Timeouts**: Both *msgQSend()* and *msgQReceive()* take timeout parameters. The timeout parameter specifies how many ticks (clock ticks per second) to wait for space to be available when sending a message and to wait for a message to be available when receiving a message. There are two constant for no timeout (WAIT\_FOREVER) and immediate timeout (NO\_WAIT).

**Urgent Messages**: The *msgQSend()* function can specify the priority of a message either as normal MSG\_PRI\_NORMAL or urgent MSG\_PRI\_URGENT. Normal priority messages are added to the tail of the message queue, while urgent priority messages are added to the head of the message queue.

**Message Queue Operations and Syntax:**

VxWorks message queues provide a single uniform interface for message queue control. Four operations are defined:

* *msgQCreate*(int maxMsgs, int maxMsgLength, int options): Allocate and initialize a message queue.
* *msgQDelete*(MSG\_Q\_ID msgQId): Terminate and free a message queue.
* *msgQSend*(MSG\_Q\_ID msgQId, char \*Buffer, UINT nBytes, int timeout, int priority): Send a message to a message queue.
* *msgQReceive*(MSG\_Q\_ID msgQId, char \*Buffer, UINT nBytes, int timeout): Receive a message from a message queue.

Communication Example:

This example uses two tasks implementing functions of a **Sender** and **Receiver**. The main program (function *message*) creates a message queue (*mqId*) and spawns sender and receiver tasks. The sender task continuously sends a message to message queue every second. The Receiver task continuously reads messages from the message queue as soon as they are available. The Sender prints out its own task Id along with the message consecutive number. The Receiver prints also its task Id with the message text (including of course the message number as created by the Sender).

**NOTE**: The function *message* terminates after spawning the two tasks, which loop infinitely - they can, however, be suspended/resumed/deleted from the shell line. Also we can spawn more Senders and Receivers (*sp Sender, sp Receiver*) as well as use the message queue identifier (*mqId*) defined in the program to send and receive messages from the command line (e.g.: *msgQSend mqId, "my message", 12*). Always check the running tasks and make note of their priority (different if you "execute" function or spawn it as a task).

**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**.Eexecute the function *message* from command line. Record and observe the output.

**A2**.Check the running tasks:

**a.** Suspend (and later resume) the sender and receiver tasks independently.

Check the status of message queue (*show mqId*). Record the output and explain what happened.

**b.** Suspend (and later resume) both tasks observing the tasks status. Check the status of message queue (*show mqId*). Record the output and explain what happened.

**A3**.With the sender(s) task(s) suspended and the receiver(s) pending, send a message from the shell command line. Show the command you used to send the message. How have message queue changed?

**A4.**Resume the Sender task and suspend the Receiver task. Show how to receive a message from the shell. Create an “urgent” message on the message queue. Repeat your command above to receive the message from the shell (HINT: you will need to create a buffer from the shell to store the received message). Explain and comment on the results?

**A5.**Experiment with spawning 2-3 receiver tasks and/or 2-3 new sender tasks. Analyze the output, task status and the queue status while suspending/resuming the tasks. Check the status of the message queue while experimenting as in the point above. Describe explicitly how do you do that and what did you learned.

**Part B:**

**B1**.Write a new program to implement Client-Server scenario with three Clients sending messages to a Server. Have the three Clients each send messages at a different rate (20, 40, 60 and ticks). The message shall be a string with the client identifier and a message number (e.g. "clientId-#"). The Server shall respond to each client with the same string appending "received at - <timestamp>" (where the <timestamp> is the time of getting the message). Each message shall be displayed three times: (a) upon creation by the client, (b) upon receiving by the server and addition of the timestamp by the server (c) upon receiving back by the client (adding second timestamp to identify the time of receiving the message back to the client). Include the code, excerpts of output, and explain the results of executing your program.

**NOTE1**: Consider spawning three threads and use only one client function. You may consider using semaphores if you need any synchronization or mutual exclusion. Also think about thread priorities. Describe how the program runs to prove that it works as specified.

**NOTE2**: See appendix for help with the timestamp implementation.

**Appendix**:

**A) *message.c***

#include <vxWorks.h> /\* Always include this as the first thing in every program \*/

#include <stdio.h> /\* we use printf \*/

#include <taskLib.h> /\* we use taskSpawn \*/

#include <sysLib.h> /\* we use sysClk... \*/

#include <msgQLib.h> /\* we use message queues \*/

/\* function prototypes \*/

void Sender(void);

void Receiver(void);

/\* defines \*/

#define MAX\_MESSAGES 100

#define MAX\_MESSAGE\_LENGTH 50

/\* globals \*/

MSG\_Q\_ID mqId;

void **message**(void) /\* function to create the message queue and two tasks \*/

{

int senderId, receiverId;

/\* create message queue \*/

if ((mqId = msgQCreate(MAX\_MESSAGES,MAX\_MESSAGE\_LENGTH,MSG\_Q\_FIFO)) == NULL)

printf("msgQCreate in failed\n");

/\* spawn the two tasks that will use the message queue \*/

if((senderId = taskSpawn("t1",110,0x100,2000,(FUNCPTR)Sender,0,0,0,0,0,0,0, 0,0,0)) == ERROR)

printf("taskSpawn taskOne failed\n");

if((receiverId = taskSpawn("t2",110,0x100,2000,(FUNCPTR)Receiver,0,0,0,0,0,0,0, 0,0,0)) == ERROR)

printf("taskSpawn taskTwo failed\n");

}

void **Sender**(void) /\* task that writes to the message queue \*/

{

char message[MAX\_MESSAGE\_LENGTH];

int i = 0;

while(1)

{

/\* create and send message \*/

sprintf(message,"message # %d from Sender %d", i, taskIdSelf());

printf("SENDER %d MESSAGE %d: \n",taskIdSelf(), i++); /\* print what is sent \*/

if((msgQSend(mqId,message,MAX\_MESSAGE\_LENGTH, WAIT\_FOREVER, MSG\_PRI\_NORMAL))

== ERROR)

printf("msgQSend in Sender failed\n");

taskDelay(sysClkRateGet()); /\* delay for one second \*/

}

}

void **Receiver**(void) /\* tasks that reads from the message queue \*/

{

char msgBuf[MAX\_MESSAGE\_LENGTH];

while(1)

{

/\* receive message \*/

if(msgQReceive(mqId,msgBuf,MAX\_MESSAGE\_LENGTH, WAIT\_FOREVER ) == ERROR)

printf("msgQReceive in Receiver failed\n");

else

printf("RECEIVER %d: %s\n",taskIdSelf(), msgBuf);

taskDelay(sysClkRateGet()/60); /\* delay for 1/60 of second (one tick) \*/ }}

**B) Timestamp**

To represent time we use structure *timespec*, defining time in seconds and nanoseconds.

struct timespec{

time\_t tv\_sec

time\_t tv\_nsec

}

tstamp;

Function *clock\_gettime()* returns the current time as the second argument (the first argument is the pre-determined constant CLOCK\_REALTIME defining real-time clock). Please refer to the VxWorks Reference Manuals for valid arguments in the above routines. To initialize time count we may use function *clock\_settime().* See the example below:

/\* *Stamp()* initializes the clock and then prints out time in sec/nsec every *step* time ticks \*/

void **Stamp**(int step)

{

tstamp.tv\_sec = 0;

tstamp.tv\_nsec = 0;

clock\_settime(CLOCK\_REALTIME, &tstamp);

while(1)

{

taskDelay(step);

clock\_gettime(CLOCK\_REALTIME, &tstamp);

printf("\ntime: %d sec %d nsec", (int) tstamp.tv\_sec, (int) tstamp.tv\_nsec);

}

}