12 Oct 2011 /TBS

FYS4220 / 9220

RT-lab no 2 - 2011

SIGNALS

1 The VxWorks signal IPC facility

VxWorks provides a software signal facility. Signals asynchronously alter the control flow of a task or process. Signals are the means by which processes/tasks are notified of the occurrence of significant events in the system, examples are hardware exceptions, POSIX RealTime signals, signals to kill a process, for notifying that a message queue has data, and so on. Each signal has a unique number, value 0 is reserved for use as the null signal. Signals can be disabled or individually enabled.

To catch and process a signal a task/process must have installed a signal handler. For VxWorks 6.2 (under Workbench) there is some difference whether the task is a kernel task or a RTP (RealTime Process). Note that the lab exercise tasks are built as kernel modules.

For more on VxWorks signals see the lecture, Workbench Help or documentation at the lab.

1.1 POSIX Queued Signals

Signals are handled differently in the kernel and in a real-time process. In the kernel the target of a signal is always a task.

The POSIX **sigqueue()** family of routines provides an alternative to the (UNIX) **kill()** family for sending signals. The important differences between the two are:

- **sigqueue()** includes an application- specified value that is sent as part of the signal, one can use this value to supply whatever is useful. This value is of type **sigval**, the signal handler finds it in the **si_value** field of one of its arguments: a structure **siginfo_t**. The use of this feature is implemented in the exercise code.
- **sigqueue()** enables the queuing of multiple signals. The **kill()** routine, by contrast, delivers only a single signal even if multiple signals arrive before the handler runs, in other words, signals may be lost.

Note however that as with all resources there is a limitation on how many signals can be stored in the **sigqueue()** buffer. If a task sends a burst of signals this limit can easily be found. This system variable can however be changed, it is left as a challenge to the students to find out more about the POSIX queued signals.

1.2 Signal handlers

Most signals are delivered asynchronously to the execution of a program. Therefore programs must be written to account for the unexpected occurrence of signals. Unlike Interrupt Service Routines (ISRs), a signal handler executes in the context of the interrupted task. Note that the task's signal handler is executed whatever the state of the task is, active or pended!

In general, signal handlers should be handled as ISR's, no routine should be called that may cause blocking. Table 3.18 of the VxWorks Application Programmer's Guide 6.2 lists those few routine that is safe. The most important ones are **logMsg()**, **msqQSend**, **semGive()** except when mutual-exclusion semaphore, **semFlush()**, a number of **taskLib** routines, **tickSet()** and **tickGet()**. **semTake()** is definitely not safe!! Anyway, it is a good idea to use **logMsg()** in all VxWorks programs since the task is identified in the output.

1.3 Communication between signal handler and task

Due to the restrictions on system calls in a handler, see sect. 1.2, actions to be taken after a signal must in general be carried out in the task. Also, like in an ISR, the time spent in a handler should be as short as possible.

But how can the signal handler notify its owner (task) that a signal has arrived? Possible methods are:

- The handler can store the signal information in global variables and the task can poll on a flag that indicates when a signal has arrived. For several reasons this is a bad solution: 1) access of shared data requires mutual exclusion and **semTake()** is not safe in a handler, and 2) polling wastes CPU time.
- The task can wait on a semaphore on which the handler execute a **semGive()** after a signal. However, in order that the task can fetch the signal information one is back to square 1) above.
- POSIX / ANSI has the (strange) **setjmp** / **longjmp** (***jmp_buf env***) facility, see lecture. The structure ***jmp_buf*** contains some extra elements that can (I think) be used for information transfer from handler to the task.
-

```
typedef struct _jmp_buf
{
        REG_SET reg;
        int     excCnt;
        int     extra[3];
} jmp_buf[1];
```

- The handler can send the information to the task over a message queue, as the **msqQSend()** is handler safe, see sect. 1.2. This is the method to be used in this exercise.

1.4 Signal system calls

The use of these functions is well illustrated in the source code. It is referred to the VxWorks documentation for additional information.

1.4.1 Signal functions arguments

The follow description of the **siginfo_t** structure is also commented into the RTlab_2 skeleton source code.

```
/* The signal handler receives the value of the signal that has triggered the handler.
 * Additional, sigqueue( )can supply an application specified value to the handler
 * of type sigval (defined in "signal.h"); the signal handler finds it in the
 * si_value field of one of its arguments, a structure siginfo_t
 *
 * The data structure is as follows:
        typedef struct siginfo
         {
                int             si_signo;
                int             si_code;
                union sigval    si_value;
         } siginfo_t;

* sigval is defined in "sigeventCommon.h"
        union sigval
        {
                int             sival_int;
                void            *sival_ptr;
         };
*/
```

How the value stored in **sival_int** is accessed see the source code.

1.4.1.1 C unions

For those of you (like me) who are not very familiar with C unions here is a practical introduction that one can also run *as is* under VxWorks.

/* A union is like a structure, except that each element shares the same memory.
Thus in this example, the **coords** and **about** members overlap.

Note that the setting of **about[2]** to 'X' CORRUPTS the float in this demonstration.

In a practical use, a variable such as un_type (provided but not used in this example) would be set up to indicate which particular use is being made of the union */

```
typedef union {
                float coords[3];
                char about[20];
} assocdata;

typedef struct {
                char *descript;
                int un_type;
                assocdata alsostuff;
} leaf;

int main() {
                leaf oak[3];
                int i;

                printf ("Hello World\n");

                for (i=0; i<3; i++) {
                                oak[i].descript = "A Greeting";
                                oak[i].un_type = 1;
```

```
                        oak[i].alsostuff.coords[0] = 3.14;
                        }
        oak[2].alsostuff.about[2] = 'X';

        for (i=0; i<3; i++) {
                        printf("%s\n",oak[i].descript);
                        printf("%5.2f\n",oak[i].alsostuff.coords[0]);
                        }
return OK;
}

/*  Sample of output from this program
Hello World
A Greeting
 3.14
A Greeting
 3.14
A Greeting
 3.39
*/
```

OK, now you know how a union works.

1.5 Waking up a task by a signal

If a task is PENDED (waiting for a resource) when a signal is sent to it, two things will take place:

- The signal handler will be executed, and
- The task is immediately taken out of the PENDED state, and will continue to execute once it gets access to the CPU.

The second point is important, because it implies that if the task is for instance waiting in **mq_receive( )** for a message or in **semTake( )** for a semaphore, an immediate return from the system call with errno = EINTR will take place without receiving neither the message nor getting the semaphore! So, if the task/process continues in good faith without testing the return status then there is trouble waiting for it.

2 *RTlab-2.c*

The source code which is the basis for the exercise is stored under **…\FYS4220\src**\ . It can also be downloaded from the FYS4220 home page.

Although the code can be compiled without errors, but with a fair number of warnings, the program system is not complete. How to complete it (in one way or another) and thereby getting the exercise approved is defined in sect. 2.2. As for the ROBOT exercise the number of lines of code to be written is few.

2.1 Structure of RTlab-2.c

The source code should be a pleasure to read, but here is a quick overview to guide the eyes. The overall structure mimics a general measurement system, with user control and monitoring, measuring tasks, and Alarm handling.

The start address is **taskControl** without parameters. It is recommended that **taskCONTROL** is started from a Target shell such that the printout is not mixed up with the output from **logMsg()** which is routed to the VxSim terminal.

The system is built with five VxWorks tasks:

- The interactive command interpreter **taskCONTROL( ),** which spawns the other tasks;
- **Alarm_task( )** which install its signal handler **SignalHandler( )** and creates a message queue over which it shall receive messages from the handler;
- **taskTEMPERATURE( )** , **taskPRESSURE( )** and **taskPOSITION ( )** . Each of them reads “data” from an array with a randomized index, and tests the value against an “error” limit. If the limit is exceeded the tasks sends a signal to **Alarm_task( )** that also includes the “data” value. In order to make the error rate more unpredictable the three tasks “reads” data at random intervals from 0 (zero) to 99 clock ticks, one tick is 16.7 ms. By decreasing the taskDelay ( ) random interval one can also test the robustness of the system.

The code includes a number of global counters that can be used to collect statistics.

2.2 What to do

1. When triggered, **SignalHandler( )** shall send the signal code and sigval (see sect. 1.4.1) parameter to **Alarm_task( )** over a message queue. Implement status test after all message queue system calls. The code for this must be added to the **SignalHandler( )** and **Alarm_task( )** skeletons.

2. Demonstrate that the number of sent and received signals is identical.

3. Try to stress the system, this can for instance be done by sending a burst of SIGRT1/SIGRT2 signals from **taskCONTROL( )** or changing the **taskDelay()** period interval in the three measuring tasks..

2.2.1 Some “help me please” info

An example of handling by **Alarm_task( )** of the received information is already implemented in the code.

According to the documentation only the VxWorks **msqQSend( )** is permitted from a signal handler. The POSIX equivalent is **mq_send( )**. (Why this difference between **msqQSend( )** and **mq_send( )** I really don’t know, because both can cause blocking if the non-blocking option has not been set.) Whatever, to profit from the message queue code in ROBOT one can use the POSIX calls. Blocking in the signal handler can be avoided by first testing for available buffers:

```
if (msgQNumMsgs((MSG_Q_ID)MQ_descriptor)  <  MQ_MSGMX-1)
```

See also sect 1.5

As for the ROBOT exercise, a significant part of the job is to study the code.

2.2.2 mq_send( ) and mq_receive( ) Reference pages

API Reference: Routines

mq_send( )

NAME

mq_send( ) - send a message to a message queue (POSIX)

SYNOPSIS

```
int mq_send
    (
    mqd_t        mqdes,   /* message queue descriptor */
    const char * pMsg,    /* message to send */
    size_t       msgLen,  /* size of message, in bytes */
    unsigned     msgPrio  /* priority of message */
    )
```

DESCRIPTION

This routine adds the message *pMsg* to the message queue *mqdes*. The *msgLen* parameter specifies the length of the message in bytes pointed to by *pMsg*. The value of *pMsg* must be less than or equal to the **mq_msgsize** attribute of the message queue, or mq_send( ) will fail.

If the message queue is not full, mq_send( ) will behave as if the message is inserted into the message queue at the position indicated by the *msgPrio* argument. A message with a higher numeric value for *msgPrio* is inserted before messages with a lower value. The value of *msgPrio* must be less than **MQ_PRIO_MAX**.

If the specified message queue is full and **O_NONBLOCK** is not set in the message queue's, mq_send( ) will block until space becomes available to queue the message, or until it is interrupted by a signal. If the message queue is full and **O_NONBLOCK** is set in the message queue's descriptions associated with *mqdes*, the message is not queued, and mq_send( ) returns **EAGAIN** error.

RETURNS

0 (OK), otherwise -1 (**ERROR**).

ERRNO

EAGAIN
O_NONBLOCK was set in the message queue description associated with *mqdes*, and the specified message queue is full

EBADF
The *mqdes* argument is not a valid message queue descriptor open for for writing

EMSGSIZE
The specified message length, *msgLen*, exceeds the message size attribute of the message queue

EINVAL
The value of *msgPrio* is greater than or equal to **MQ_PRIO_MAX** the *pMsg* pointer is invalid

EINTR
The request has been interrupted by a signal

SEE ALSO

mqPxLib, mq_receive( )

API Reference: Routines

mq_receive( )

NAME

mq_receive( ) - receive a message from a message queue (POSIX)

SYNOPSIS

```
ssize_t mq_receive
    (
    mqd_t      mqdes,     /* message queue descriptor */
    char     * pMsg,      /* buffer to receive message */
    size_t     msgLen,    /* size of buffer, in bytes */
    unsigned * pMsgPrio   /* if not NULL, priority of message */
    )
```

DESCRIPTION

This routine receives the oldest of the highest priority message from the message queue specified by *mqdes*. If the size of the buffer in bytes, specified by the *msgLen* argument, is less than the **mq_msgsize** attribute of the message queue, mq_receive( ) will fail and return an error. Otherwise, the selected message is removed from the queue and copied to *pMsg*.

If *pMsgPrio* is not **NULL**, the priority of the selected message will be stored in *pMsgPrio*.

If the message queue is empty and **O_NONBLOCK** is not set in the message queue's description, mq_receive( ) will block until a message is added to the message queue, or until it is interrupted by a signal. If more than one task is waiting to receive a message when a message arrives at an empty queue, the task of highest priority that has been waiting the longest will be selected to receive the message. If the specified message queue is empty and **O_NONBLOCK** is set in the message queue's description, no message is removed from the queue, and mq_receive( ) returns an error.

RETURNS

The length of the selected message in bytes, otherwise -1 (**ERROR**).

ERRNO

EAGAIN
O_NONBLOCK was set in the message queue description associated with *mqdes*, and the specified message queue is empty.

EBADF
The *mqdes* argument is not a valid message queue descriptor open for for reading.

EMSGSIZE
The specified message buffer size, *msgLen*, is less than the message size attribute of the message queue.

EINVAL
The *pMsg* pointer is invalid.

EINTR
Signal received while blocking on the message queue.

SEE ALSO

mqPxLib, mq_send( )

2.3 Output from a perfect system

See Figure 1 and Figure 2

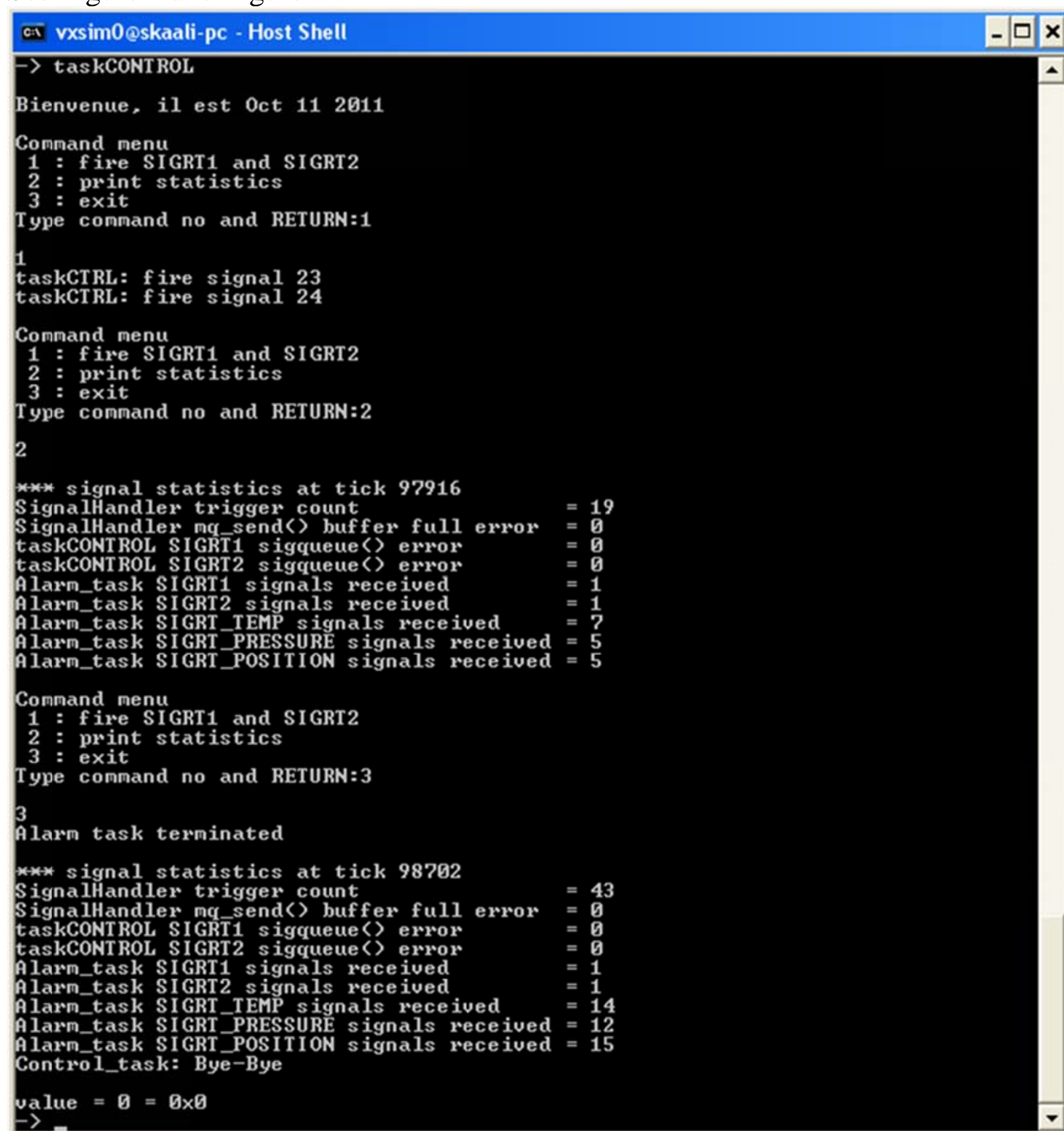

Figure 1 Shell output from taskCONTROL( )

```
VxSim0
->
->
-> 0x11700d78 (tAlarm): ALARM at clock tick 97240: position = -2
0x11700d78 (tAlarm): ALARM at clock tick 97248: pressure = 91
0x11700d78 (tAlarm): ALARM at clock tick 97258: position = -6
0x11700d78 (tAlarm): ALARM at clock tick 97313: position = -3
0x11700d78 (tAlarm): ALARM at clock tick 97341: pressure = 95
0x11700d78 (tAlarm): ALARM at clock tick 97407: temperature = 64
0x11700d78 (tAlarm): ALARM at clock tick 97428: pressure = 95
0x11700d78 (tAlarm): ALARM at clock tick 97473: temperature = 68
0x11700d78 (tAlarm): ALARM at clock tick 97494: pressure = 94
0x11700d78 (tAlarm): ALARM at clock tick 97511: position = -1
0x11700d78 (tAlarm): ALARM at clock tick 97540: pressure = 91
0x11700d78 (tAlarm): ALARM at clock tick 97558: temperature = 64
0x11700d78 (tAlarm): ALARM at clock tick 97672: temperature = 52
0x11700d78 (tAlarm): ALARM at clock tick 97728: temperature = 52
0x11700d78 (tAlarm): ALARM at clock tick 97751: position = -6
0x11700d78 (tAlarm): ALARM at clock tick 97821: temperature = 52
0x11700d78 (tAlarm): ALARM at clock tick 97901: temperature = 68
0x11700d78 (tAlarm): ALARM at clock tick 97975: temperature = 77
0x11700d78 (tAlarm): ALARM at clock tick 97987: position = -2
0x11700d78 (tAlarm): ALARM at clock tick 98033: temperature = 68
0x11700d78 (tAlarm): ALARM at clock tick 98057: pressure = 91
0x11700d78 (tAlarm): ALARM at clock tick 98075: position = -6
0x11700d78 (tAlarm): ALARM at clock tick 98095: temperature = 68
0x11700d78 (tAlarm): ALARM at clock tick 98211: pressure = 91
0x11700d78 (tAlarm): ALARM at clock tick 98351: temperature = 64
0x11700d78 (tAlarm): ALARM at clock tick 98380: pressure = 92
0x11700d78 (tAlarm): ALARM at clock tick 98413: temperature = 77
0x11700d78 (tAlarm): ALARM at clock tick 98419: position = -1
0x11700d78 (tAlarm): ALARM at clock tick 98461: pressure = 95
0x11700d78 (tAlarm): ALARM at clock tick 98487: temperature = 68
0x11700d78 (tAlarm): ALARM at clock tick 98491: pressure = 91
0x11700d78 (tAlarm): ALARM at clock tick 98511: position = -2
0x11700d78 (tAlarm): ALARM at clock tick 98559: position = -1
0x11700d78 (tAlarm): ALARM at clock tick 98571: pressure = 95
0x11700d78 (tAlarm): ALARM at clock tick 98594: position = -3
0x11700d78 (tAlarm): ALARM at clock tick 98633: pressure = 95
0x11700d78 (tAlarm): ALARM at clock tick 98634: position = -1
0x11700d78 (tAlarm): ALARM at clock tick 98648: temperature = 77
0x11700d78 (tAlarm): ALARM at clock tick 98652: position = -2
0x11700d78 (tAlarm): ALARM at clock tick 98659: position = -2
0x11700d78 (tAlarm): ALARM at clock tick 98667: position = -2
0x11700d78 (tAlarm): Alarm task terminated

->
```

Figure 2 Terminal output from Alarm_task()