Extrait du man VxWorks

14 Février 2002

NAME

taskSpawn() - spawn a task

SYNOPSIS

```
int taskSpawn
    (
    char     *name,      /* name of new task (stored at pStackBase)   */
    int      priority,   /* priority of new task                      */
    int      options,    /* task option word                          */
    int      stackSize,  /* size (bytes) of stack needed plus name    */
    FUNCPTR  entryPt,    /* entry point of new task                   */
    int      arg1,       /* 1st of 10 req'd task args to pass to func */
    int      arg2,
    int      arg3,
    int      arg4,
    int      arg5,
    int      arg6,
    int      arg7,
    int      arg8,
    int      arg9,
    int      arg10
    )
```

DESCRIPTION

This routine creates and activates a new task with a specified priority and options and returns a system-assigned ID. See taskInit() and taskActivate() for the building blocks of this routine.

A task may be assigned a name as a debugging aid. This name will appear in displays generated by various system information facilities such as i(). The name may be of arbitrary length and content, but the current VxWorks convention is to limit task names to ten characters and prefix them with a "t". If <name> is specified as NULL, an ASCII name will be assigned to the task of the form "t<n>" where <n> is an integer which increments as new tasks are spawned.

The only resource allocated to a spawned task is a stack of a specified size <stackSize>, which is allocated from the system memory partition. Stack size should be an even integer. A task control block (TCB) is carved from the stack, as well as any memory required by the task name. The remaining memory is the task's stack and every byte is filled with the value 0xEE for the checkStack() facility. See the manual entry for checkStack() for stack-size checking aids.

The entry address <entryPt> is the address of the "main" routine of the task. The routine will be called once the C environment has been set up. The specified routine will be called with the ten given arguments. Should the specified main routine return, a call to exit() will

automatically be made.

Note that ten (and only ten) arguments must be passed for the spawned function.

Bits in the options argument may be set to run with the following modes: execute with floating-point coprocessor support. include private environment support (see envLib). do not fill the stack for use by checkstack(). do not allow breakpoint debugging.

See the definitions in taskLib.h.

RETURNS

The task ID, or ERROR if memory is insufficient or the task cannot be created.

NAME

taskDelay() - delay a task from executing

SYNOPSIS

```
STATUS taskDelay
    (
    int  ticks  /* number of ticks to delay task */
    )
```

DESCRIPTION

This routine causes the calling task to relinquish the CPU for the duration specified (in ticks). This is commonly referred to as manual rescheduling, but it is also useful when waiting for some external condition that does not have an interrupt associated with it.

If the calling task receives a signal that is not being blocked or ignored, taskDelay() returns ERROR and sets 'errno' to EINTR after the signal handler is run.

RETURNS

OK, or ERROR if called from interrupt level or if the calling task receives a signal that is not blocked or ignored.

NAME

tickGet() - get the value of the kernel's tick counter

SYNOPSIS

ULONG tickGet (void)

DESCRIPTION

This routine returns the current value of the tick counter. This value is set to zero at startup, incremented by tickAnnounce(), and can be changed using tickSet().

RETURNS

The most recent tickSet() value, plus all tickAnnounce() calls since.

```
NAME
       semCCreate() - create and initialize a counting semaphore

SYNOPSIS
       SEM_ID semCCreate
           (
           int  options,      /* semaphore option modes */
           int  initialCount  /* initial count          */
           )

DESCRIPTION
       This   routine   allocates   and  initializes  a  counting
       semaphore.  The semaphore is initialized to the  specified
       initial count.

       The  <options>  parameter  specifies the queuing style for
       blocked tasks.  Tasks may be queued on a priority basis or
       a  first-in-first-out basis.  These options are SEM_Q_PRI
       ORITY and SEM_Q_FIFO, respectively.

RETURNS
       The semaphore ID, or NULL if memory cannot be allocated.
```

NAME

semMCreate() - create and initialize a mutual-exclusion semaphore

SYNOPSIS

```
SEM_ID semMCreate
    (
    int  options  /* mutex semaphore options */
    )
```

DESCRIPTION

This routine allocates and initializes a mutual-exclusion semaphore. The semaphore state is initialized to full.

Semaphore options include the following: Queue pended tasks on the basis of their priority. Queue pended tasks on a first-in-first-out basis. Protect a task that owns the semaphore from unexpected deletion. This option enables an implicit taskSafe() for each semTake(), and an implicit taskUnsafe() for each semGive(). Protect the system from priority inversion. With this option, the task owning the semaphore will execute at the highest pri ority of the tasks pended on the semaphore, if it is higher than its current priority. This option must be accompanied by the SEM_Q_PRIORITY queuing mode.

RETURNS

The semaphore ID, or NULL if memory cannot be allocated.

```
NAME
       semGive() - give a semaphore

SYNOPSIS
       STATUS semGive
           (
           SEM_ID  semId  /* semaphore ID to give */
           )

DESCRIPTION
       This  routine  performs  the give operation on a specified
       semaphore.  Depending on the type of semaphore, the  state
       of the semaphore and of the pending tasks may be affected.
       The behavior  of  semGive()  is  discussed  fully  in  the
       library  description  of the specific semaphore type being
       used.

RETURNS
       OK, or ERROR if the semaphore ID is invalid.
```

NAME

semTake() - take a semaphore

SYNOPSIS

```
STATUS semTake
    (
    SEM_ID  semId,   /* semaphore ID to take */
    int     timeout  /* timeout in ticks     */
    )
```

DESCRIPTION

This routine performs the take operation on a specified semaphore. Depending on the type of semaphore, the state of the semaphore and the calling task may be affected. The behavior of semTake() is discussed fully in the library description of the specific semaphore type being used.

A timeout in ticks may be specified. If a task times out, semTake() will return ERROR. Timeouts of WAIT_FOREVER and NO_WAIT indicate to wait indefinitely or not to wait at all.

The semTake() routine is not callable from interrupt ser vice routines.

RETURNS

OK, or ERROR if the semaphore ID is invalid or the task timed out.