Extrait du manuel VxWorks

(http://spacegrant.colorado.edu/~dixonc/vxworks/docs/vxworks/ref/)

fichiers datant du 6 Mars 2001

taskSpawn( )

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:

VX_FP_TASK (0x0008)

execute with floating-point coprocessor support.

VX_PRIVATE_ENV (0x0080)

include private environment support (see **envLib**).

VX_NO_STACK_FILL (0x0100)

do not fill the stack for use by ***checkStack*( )**.

VX_UNBREAKABLE (0x0002)

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.

ERRNO

S_intLib_NOT_ISR_CALLABLE, S_objLib_OBJ_ID_ERROR, S_smObjLib_NOT_INITIALIZED, S_memLib_NOT_ENOUGH_MEMORY, S_memLib_BLOCK_ERROR

---

taskDelay( )

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.

ERRNO

S_intLib_NOT_ISR_CALLABLE, EINTR

---

tickGet( )

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.

---

semCCreate( )

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_PRIORITY** (0x1) and **SEM_Q_FIFO** (0x0), respectively.

RETURNS

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

---

semMCreate( )

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:

SEM_Q_PRIORITY (0x1)

Queue pended tasks on the basis of their priority.

SEM_Q_FIFO (0x0)

Queue pended tasks on a first-in-first-out basis.

SEM_DELETE_SAFE (0x4)

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*( )**.

SEM_INVERSION_SAFE (0x8)

Protect the system from priority inversion. With this option, the task owning the semaphore will execute at the highest priority 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.

---

semGive( )

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.

ERRNO

S_intLib_NOT_ISR_CALLABLE, S_objLib_OBJ_ID_ERROR,
S_semLib_INVALID_OPERATION

---

semTake( )

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** (-1) and **NO_WAIT** (0) indicate to wait indefinitely or not to wait at all.

When ***semTake*( )** returns due to timeout, it sets the errno to S_objLib_OBJ_TIMEOUT (defined in **objLib.h**).

The ***semTake*( )** routine is not callable from interrupt service routines.

RETURNS

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

ERRNO

S_intLib_NOT_ISR_CALLABLE, S_objLib_OBJ_ID_ERROR,
S_objLib_OBJ_UNAVAILABLE

---

taskIdSelf( )

NAME

***taskIdSelf*( ) -** get the task ID of a running task

SYNOPSIS

```
int taskIdSelf (void)
```

DESCRIPTION

This routine gets the task ID of the calling task. The task ID will be invalid if called at interrupt level.

RETURNS

The task ID of the calling task.

taskName( )

NAME

***taskName*( ) -** get the name associated with a task ID

SYNOPSIS

```
char *taskName
    (
    int tid /* ID of task whose name is to be found */
    )
```

DESCRIPTION

This routine returns a pointer to the name of a task of a specified ID, if the task has a name. If the task has no name, it returns an empty string.

RETURNS

A pointer to the task name, or NULL if the task ID is invalid.

---

taskLock( )

NAME

***taskLock*( ) -** disable task rescheduling

SYNOPSIS

```
STATUS taskLock (void)
```

DESCRIPTION

This routine disables task context switching. The task that calls this routine will be the only task that is allowed to execute, unless the task explicitly gives up the CPU by making itself no longer ready. Typically this call is paired with ***taskUnlock*( )**; together they surround a critical section of code. These preemption locks are implemented with a counting variable that allows nested preemption locks. Preemption will not be unlocked until ***taskUnlock*( )** has been called as many times as ***taskLock*( )**.

This routine does not lock out interrupts; use ***intLock*( )** to lock out interrupts.

A ***taskLock*( )** is preferable to ***intLock*( )** as a means of mutual exclusion, because interrupt lock-outs add interrupt latency to the system.

A ***semTake*( )** is preferable to ***taskLock*( )** as a means of mutual exclusion, because

preemption lock-outs add preemptive latency to the system.

The ***taskLock*( )** routine is not callable from interrupt service routines.

RETURNS

OK or ERROR.

ERRNO

S_objLib_OBJ_ID_ERROR, S_intLib_NOT_ISR_CALLABLE

---

taskUnlock( )

NAME

***taskUnlock*( ) -** enable task rescheduling

SYNOPSIS

```
STATUS taskUnlock (void)
```

DESCRIPTION

This routine decrements the preemption lock count. Typically this call is paired with ***taskLock*( )** and concludes a critical section of code. Preemption will not be unlocked until ***taskUnlock*( )** has been called as many times as ***taskLock*( )**. When the lock count is decremented to zero, any tasks that were eligible to preempt the current task will execute.

The ***taskUnlock*( )** routine is not callable from interrupt service routines.

RETURNS

OK or ERROR.

ERRNO

S_intLib_NOT_ISR_CALLABLE

---

sysClkRateGet( )

NAME

***sysClkRateGet*( ) -** get the system clock rate

SYNOPSIS

```
int sysClkRateGet (void)
```

DESCRIPTION

This routine returns the system clock rate.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.*

RETURNS

The number of ticks per second of the system clock.

---

nanosleep( )

NAME

***nanosleep*( )** - suspend the current task until the time interval elapses (POSIX)

SYNOPSIS

```
int nanosleep
    (
    const struct timespec * rqtp, /* time to delay */
    struct timespec *       rmtp  /* premature wakeup (NULL=no result) */
    )
```

DESCRIPTION

This routine suspends the current task for a specified time *rqtp* or until a signal or event notification is made.

The suspension may be longer than requested due to the rounding up of the request to the timer's resolution or to other scheduling activities (e.g., a higher priority task intervenes).

If *rmtp* is non-NULL, the **timespec** structure is updated to contain the amount of time remaining. If *rmtp* is NULL, the remaining time is not returned. The *rqtp* parameter is greater than 0 or less than or equal to 1,000,000,000.

RETURNS

0 (OK), or -1 (ERROR) if the routine is interrupted by a signal or an asynchronous event notification, or *rqtp* is invalid.

ERRNO

EINVAL, EINTR

msgQCreate( )

NAME

***msgQCreate*( ) -** create and initialize a message queue

SYNOPSIS

```
MSG_Q_ID msgQCreate
    (
    int maxMsgs,      /* max messages that can be queued */
    int maxMsgLength, /* max bytes in a message */
    int options       /* message queue options */
    )
```

DESCRIPTION

This routine creates a message queue capable of holding up to *maxMsgs* messages, each up to *maxMsgLength* bytes long. The routine returns a message queue ID used to identify the created message queue in all subsequent calls to routines in this library. The queue can be created with the following options:

MSG_Q_FIFO (0x00)

queue pended tasks in FIFO order.

MSG_Q_PRIORITY (0x01)

queue pended tasks in priority order.

RETURNS

MSG_Q_ID, or NULL if error.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY, S_intLib_NOT_ISR_CALLABLE

---

msgQSend( )

NAME

***msgQSend*( ) -** send a message to a message queue

SYNOPSIS

```
STATUS msgQSend
```

```
(
MSG_Q_ID msgQId,  /* message queue on which to send */
char *   buffer,  /* message to send */
UINT     nBytes,  /* length of message */
int      timeout, /* ticks to wait */
int      priority /* MSG_PRI_NORMAL or MSG_PRI_URGENT */
)
```

DESCRIPTION

This routine sends the message in *buffer* of length *nBytes* to the message queue *msgQId*. If any tasks are already waiting to receive messages on the queue, the message will immediately be delivered to the first waiting task. If no task is waiting to receive messages, the message is saved in the message queue.

The *timeout* parameter specifies the number of ticks to wait for free space if the message queue is full. The *timeout* parameter can also have the following special values:

NO_WAIT (0)

return immediately, even if the message has not been sent.

WAIT_FOREVER (-1)

never time out.

The *priority* parameter specifies the priority of the message being sent. The possible values are:

MSG_PRI_NORMAL (0)

normal priority; add the message to the tail of the list of queued messages.

MSG_PRI_URGENT (1)

urgent priority; add the message to the head of the list of queued messages.

USE BY INTERRUPT SERVICE ROUTINES

This routine can be called by interrupt service routines as well as by tasks. This is one of the primary means of communication between an interrupt service routine and a task. When called from an interrupt service routine, *timeout* must be **NO_WAIT**.

RETURNS

OK or ERROR.

ERRNO

S_distLib_NOT_INITIALIZED, S_objLib_OBJ_ID_ERROR,
S_objLib_OBJ_DELETED, S_objLib_OBJ_UNAVAILABLE,
S_objLib_OBJ_TIMEOUT, S_msgQLib_INVALID_MSG_LENGTH,
S_msgQLib_NON_ZERO_TIMEOUT_AT_INT_LEVEL

msgQReceive( )

NAME

***msgQReceive*( ) -** receive a message from a message queue

SYNOPSIS

```
int msgQReceive
    (
    MSG_Q_ID msgQId,    /* message queue from which to receive */
    char *   buffer,    /* buffer to receive message */
    UINT     maxNBytes, /* length of buffer */
    int      timeout    /* ticks to wait */
    )
```

DESCRIPTION

This routine receives a message from the message queue *msgQId*. The received message is copied into the specified *buffer*, which is *maxNBytes* in length. If the message is longer than *maxNBytes*, the remainder of the message is discarded (no error indication is returned).

The *timeout* parameter specifies the number of ticks to wait for a message to be sent to the queue, if no message is available when ***msgQReceive*( )** is called. The *timeout* parameter can also have the following special values:

NO_WAIT (0)

return immediately, even if the message has not been sent.

WAIT_FOREVER (-1)

never time out.

WARNING

This routine must not be called by interrupt service routines.

RETURNS

The number of bytes copied to *buffer*, or ERROR.

ERRNO

S_distLib_NOT_INITIALIZED, S_smObjLib_NOT_INITIALIZED,
S_objLib_OBJ_ID_ERROR, S_objLib_OBJ_DELETED,
S_objLib_OBJ_UNAVAILABLE, S_objLib_OBJ_TIMEOUT,
S_msgQLib_INVALID_MSG_LENGTH