Real-Time Systems Project Assignment 1

Part 1

The assignment asks you to look into the task structures of vxWork, eCos, and µITRON. You will need to know how a task (or thread) is represented and what can be done to control its execution.

1.- About eCos and µ ITRON

Confusion about the relation and implementations of the eCos Operating System and µ ITRON may appear. Before starting the task (thread) comparison between the three OS proposed, it is necessary to clarify this relationship, and this section will attempt to do so.

ITRON (Industrial - The Real-time Operating system Nucleus) is a real-time, multitasking OS specification intended for use in industrial embedded systems. µ ITRON specification is a highly condensed version of ITRON specification, which was initially implemented on 8-bit microcontrollers (MCUs), but that lately has also been ported to 16 and 32-bit microprocessors.

Several revisions of the µITRON specification exist. In this release (the manual release we are using), eCos supports the µITRON version 3.02 specification. This basically means that the eCos kernel implements the functionality used by the µITRON compatibility subsystem. This includes several task management functions (as well as other level functions such as synchronization and communication, time management, etc) and also error handling related issues. Most of the native thread functions in eCos system can indeed be translated to µITRON functions. This parallelism will be described afterwards.

Like all parts of the eCos system, the detailed semantics of the µITRON layer are dependent on its configuration and the configuration of other components that it uses. The µITRON configuration options are all defined in an header file and can be configured using specific tools or editing certain files.

2.- Task States and transition

Kernel maintains the current state of each task in the system and provides function calls to change the task from one state to another. The three OS specification define that when a thread is created and its structures built, its initial state is set to inactive until is specifically activated. Anyway, although the concepts of active and inactive are very similar in all the OS, the state names vary and may lead to confusion (even with eCos and µITRON). State names of the different OS and its meaning are exposed following:

µITRON specifies 5 different states and 3 sub-states, from which only 3 are compulsory and the rest are implementation dependent. This states are:

1. RUN: Task is running
2. READY: This state indicates that the task is ready to execute, but cannot because a task of higher priority (sometimes same priority) is already executing.
3. (General) Wait: This state indicates that the task cannot execute because conditions necessary for execution have not yet been met. General wait state can be further classified into three types:
    a. [3.1] (Specific) WAIT: The task suspends execution due to a system call it has issued itself.
    b. [3.2] SUSPEND: A task has been forcibly made to suspend execution by another task.
    c. [3.3] WAIT-SUSPEND: Conditions of both WAIT and SUSPEND state apply.
4. DORMANT: This state indicates the task is not yet executing or has already exited. Newly created tasks always begin in this state
5. NON-EXISTENT: This indicates a virtual state where the task in question does not exist on the system because it has not yet been created or has already been deleted.

Only run, wait and ready states are compulsory.

Ecos follows µITRON specification implementing 5 of the 7 states stated in it and adding a 6th extended state:

1. RUNNING: A thread holding this state is runnable or running. Note that run and ready states has been fused in only one state
2. SLEEPING: This state is equivalent to the wait state in µITRON specification
3. COUNTSLEEP: This state is equivalent to the wait state in µITRON specification, with the difference that a time to wake up can be specified. Although µITRON (the same as eCos) provides a function to temporally sleep a task (*tsl_tsk* and *cyg_thread_delay* respectively), µITRON does not consider it as a separate state.
4. SUSPENDED: This state is equivalent to the SUSPEND state in µITRON specification. Note that each task in eCos holds a counter of suspended calls, so a task can not enter suspended state when the number of this calls is higher than the number of resume calls.
5. CREATING: Thread is being created. This is equivalent to the DORMANT state of µITRON specification
6. EXITED: Thread has exited. This is equivalent to the NON-EXISTENT state in µITRON specification.

VxWorks defines also 5 main states plus 5 composed states. The 5 main ones are:

1. READY: RUN state is not specifically considered, and it is stated that the highest priority ready task is executing. In general, READY state is the state of a task that is not waiting for any resource but the CPU, and is equivalent to READY and RUNNING states is µITRON and eCos respectively.
2. PEND: Is the sate of a task that is blocked due to the unavailability of some resource. This is equivalent to the SUSPEND and SUSPENDED states in µITRON and eCos respectively, although there is not an specific function to change the state of a task to PEND as it changes automatically to this state when is blocked (such as when waiting in a semaphore).
3. DELAY: Task is asleep for some duration. Is equivalent to the WAIT and SLEEPING/COUNTSLEEP states in µITRON and eCos respectively.
4. SUSPEND: Although this state is specified and can drive to confusion due to the existence of states with the same name in µITRON and eCos, is only used in VxWorks for debugging and starting tasks. However, the fact that this is the only way to make a task to suspend specifically makes it closer to the other SUSPEND states named previously.

The following table summarizes the control functions used by the OS to change the state of the tasks:

	µITRON	Native eCos	VxWorks
Suspend	sus_tsk	cyg_thread_suspend	taskSuspend (SUSPEND)
Resume	rsm_tsk frsm_tsk (forced)	cyg_thread_resume cyg_thread_release (forced)	taskResume
Sleep	slp_task tslp_task (counter)	cyg_thread_delay (counter)	taskDelay nanosleep
Wake up	wup_task rel_wai	Counter expires	Counter expires

Additional state related functions:

	µITRON	Native eCos	VxWorks
Cancel Wakeup request	can_wup	No equivalent	No equivalent
Rotate tasks on ready queue	rot_rqd	cyg_thread_yield	sched_yield
Task Restart	No equivalent	reinitialize	TaskRestart

3.- Task creation and termination

Create and delete task functions allows application code and OS packages to create and destroy threads. In many applications this only happens during system initialization and all required data is allocated statically. However additional threads can be created at any time, if necessary.

For µITRON specification, for example, *cre_task* and *del_tsk* specify creation and deletion of tasks with dynamic memory allocation and memory space release. However, in eCos implementation of the µITRON specification, because of the static initialization facilities provided for system objects, a task is allocated stack space statically in the configuration. So while tasks can be created and deleted, the same stack space is used for that task each time. Thus the stack size requested in cre_tsk() is checked for being less than that which was statically allocated, and otherwise ignored. This ensures that the new task will have enough stack to run. For this reason del_tsk() does not in any sense free the memory that was in use for the task's stack. Moreover, Dynamic thread stack allocation is only provided if there is an implementation of malloc() configured (i.e. a package implements the CYGINT_MEMALLOC _MALLOC_ALLOCATORS interface). If there is no malloc() available, then the thread creator must supply a stack.

Following the specific functions and its explanation are specified:
µITRON specifies the *cre_tsk* and *del_tsk / ter_tsk* functions.

cre_tsk: This system call creates the task specified by the parameter *tskid*. Specifically, a TCB (Task Control Block) is allocated for the task to be created, and initialised according to accompanying parameter values specified. A stack area is also allocated for the task based on the parameter stksz (stack size). Implementation dependent information might be included together with the rest of the paramenters.

del_tsk: This system call deletes the task specified by tskid. Specifically, it changes the state of the task specified by tskid from DORMANT into NON-EXISTENT (a virtual state not existing on the system), and then clears the TCB and releases stack. An error results if this system call is used on a task which is not DORMANT

ter_tsk: Same as del_tsk but the results are applied to a task different than the one issuing the command

Related functions:

ext_tsk: This system call causes the issuing task to exit, changing the state of the task into the DORMANT state.

exd_tsk: The issuing task exists and deletes (ext_tsk + del_tsk)

sta_tsk: When a new task is created, its state is set automatically to DORMANT. This call changes the state of a task specified from DORMANT into RUN/READY

eCos specifies the *cyg_thread_create* and *cyg_thread_delete / cyg_thread_kill* functions.

cyg_thread_create: This function is equivalent to the *cre_tsk* from the µITRON specification. Additionally a name and a handler are also specified for a certain task. The name argument is used primarily for debugging purposes, making it easier to keep track of which cyg_thread structure is associated with which application-level thread. The kernel configuration option CYG-VAR_KERNEL_THREADS_NAME controls whether or not this name is actually used. Also on creation each thread is assigned a unique handle besides its ID, and this will be stored in the location pointed at by the handle argument. Some functions require passing the thread's handler pointer as a parameter rather than its ID

cyg_thread_delete: This function is equivalent to the *del_tsk* from the µITRON specification. If it is issued over a task that is not on the EXITED state, the task is exited and then deleted (contrary to the µITRON specification where it has to be explicitly called)

cyg_thread_kill: This function is equivalent to the *ter_tsk* from the µITRON specification.

Related functions:

cyg_thread_exit: This system call causes the issuing task to exit, changing the state of the task into the EXITED state.

cyg_thread_resume: Aside from resuming a suspended task, this function is used in a equivalent way as the sta_tsk is used in the µITRON specification.

cyg_thread_add_destructor: These functions are provided for cases when an application requires a function to be automatically called when a thread exits. This is often useful when, for example, freeing up resources allocated by the thread. This support must be enabled with the configuration option CYGPKG_KERNEL_ THREADS_DESTRUCTORS.

cyg_thread_add_destructor: Delete a previously created destructor.

VxWorks specifies the *taskSpawn* and *exit / taskDelete* funtions.

taskSpawn: Also a *name* parameter can be specified. Moreover, 10 additional arguments for task initialisation can be used. Furthermore, several task options, such as using floating-point coprocessor or disabling breakpoints can be specified

exit: Similar to previous delete functions, only task stacks and task control block are freed, while the memory allocated but the task during its execution is not freed.

taskDelete: Terminate a given task.

Related functions:

taskInit and *taskActivate*: Although this functions are called by *taskSpawn*, they can be called manually when need more control over allocation and activation.

taskSafe and *taskUnsafe*: The routine *taskSafe* protects a task from deletion by other tasks. This protection is often needed when a task executes in a critical region or engages a critical resource.

TaskDeleteHookAdd and *taskDeleteHookDelete:* Equivalent functionality to eCos destructors specified previously.

TaskCreateHookAdd and *taskCreateHookDelete:* The same way eCos provides destructors when a task is deleted, VxWorks also provides support for routine call when a task is created

TaskSwitchHookAdd and *taskSwitchHookDelete:* The same way eCos provides destructors when a task is deleted, VxWorks also provides support for routine call on task switch

4.- Task information request

Functions to obtain task information are useful mainly during debugging. Following several of these functions are shown. Note that the functions regarding priority information are explained in section 5.

µITRON specifies the following general functions. Note that the information stored in a task in some times implementation dependent, thus µITRON provide mostly a general function:

get_tid: This system call gets the ID of the issuing task.

ref_tsk: This system call refers to the state of the task specified by tskid, and returns its current priority (tskpri), its task state (tskstat), and its extended information (exinf). Depending on the implementation, the following additional information can also be referenced in addition to exinf, tskpri and tskstat.

tskwait	Reason for wait
wid	Wait object ID
wupcnt	Number of queued wakeup requests
suscnt	Number of nested SUSPEND requests

tskatr	Task attributes
task	Task starting address
itskpri	Initial task priority
stksz	Stack size

eCos specifies also a set of functions to request task information. This functions are not as general as the µITRON ones, although they implement an specific part of them.

cyg_thread_self: This system call gets the handler of the issuing task.
cyg_thread_idle_thread: Same from system idle thread
cyg_thread_get_stack_base and *cyg_thread_stack_size***:** Get stack base address and size of a thread.
cyg_thread_measure_stack_usage: Only available if configured, this function returns the number of bytes used so far by the thread.
cyg_thread_get_next: This function is used to enumerate all the current threads in the system. It should be called initially with the locations pointed to by thread and id set to zero. On return these will be set to the handle and ID of the first thread. On subsequent calls, these parameters should be left set to the values returned by the previous call. The handle and ID of the next thread in the system will be installed each time, until a false return value indicates the end of the list.
cyg_thread_get_info: Similar to the macro function ref_tsk from µITRON, it returns general information about the requested thread. Please se the data structures in section 6 about the cyg_thread_info structure.
*cyg_thread_find:*Returns the handler for a thread given its ID.

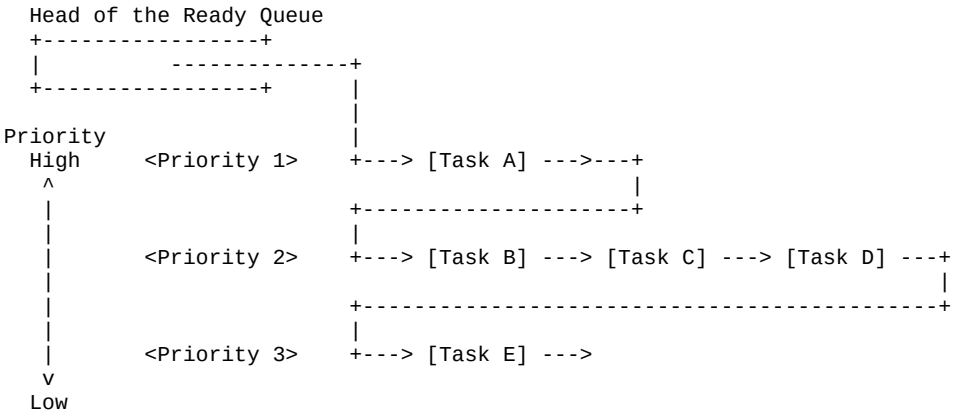
VxWorks specifies also a set of functions to request task information. Some of this functions are similar to other OS, and the mainly difference stays in the different information specified at task creation time:

taskOptionsGet and *taskOptionsSet:* This functions fill the functionality required by the *Options* field specified at the *taskSpawn* function.
*taskIdListGet:*Fills an array with the IDs of the active tasks. This functionality is accomplished in the eCos OS with the *cyg_thread_get_next* funtion
taskRegsGet and *tasksRegsSet:* Examine and set a certain task registers. VxWorks provide processor registers in its TCB structure. Contrary, the eCos OS provide this information in separate files according to the specific HAL. Ecos doesn't provide specific functions to manage this registers.
taskIsSuspended and *taskIsReady:* Checks if a task is supended or ready. This functionality is not specifically provided by the other OS, but general information about the current task state is indeed available.
taskTcb: µITRON and eCos don't provide specifically structures called task control blocks, but several information structures that can be gathered with different command requests. However, VxWorks provides a unified TCB structure, which can be accessed through a pointer obtained with this function.

5.- Scheduling

Although several task design factors, such as exception handling or inter-task communication could be taken into consideration for this report, is not possible to fit detailed information about them. However, trying to reach a compromise in this RTOS analysis, it has been decided to provide some scheduling information. Moreover, scheduling decisions, such as the ones taken from the inclusion of priorities, affect directly the task design and have constituted important factor for its inclusion in this writing. Without more delay, lets the scheduling designs of the different OS considered:

Under ***µITRON*** specification, task scheduling is conducted based on task priority. If there are some tasks of the same priority scheduling is conducted on "first come, first served" (FCFS) basis. The µITRON specification text file, edit by Ken Sakamura, provides the following picture:



Conceptually, the ready queue includes not only tasks in the READY state, but the task in the RUN state as well. Moreover, µITRON also provides pre-emption in the case that a higher priority task enters the READY state when a lower priority task is executing. However, the task that is pre-empted will not loose its position in the ready queue and will be re-scheduled when the current task finished its execution.

µITRON provides the following functions related with task priority:

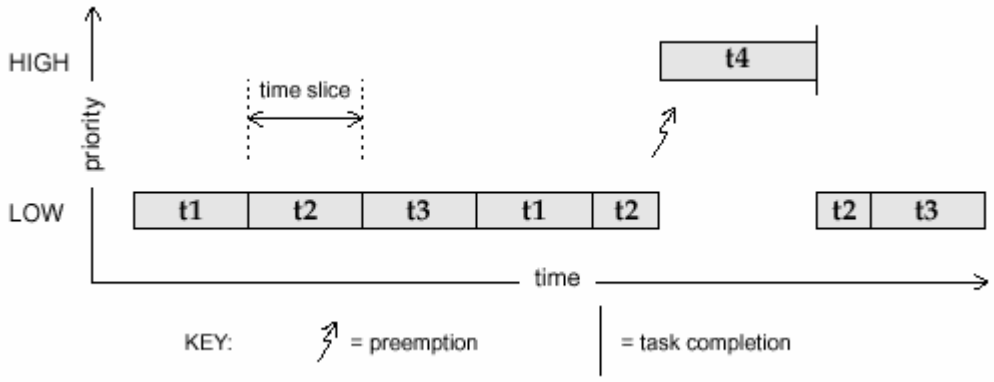
rot_rdq: This function rotates tasks on the ready queue for the same priority.
chg_pri: This function allows to change dynamically the priority of a task.
ref_tsk: Offers task information, including task current priority

As an implementation of µITRON specification, ***eCos*** also provides the multilevel queue priority scheduling. However, the possibility of a single level priority scheduling, called the bitmap scheduler, which only allows one thread per priority (32 priority levels), causing a number of limitations (the most obvious one being the existence of maximum 32 threads in the system simultaneously).

ECos provides the following functions related with task priority:

cyg_thread_get_priosity and *cyg_thread_get_current_priosity:* This function gets the priority of a given ask.
cyg_thread_set_priosity: This function allows to change dynamically the priority of a task, the same way that in the µITRON specification is provided with the *chg_pri* function.

VxWorks follows the same design as eCos, providing a priority-based pre-emptive scheduling. However, the offered alternative is also a multilevel priority scheduling but based in a round-robin policy for the tasks running in the same priority. This way the scheduler achieves fairness within tasks of the same priority by enabling a time slice (which can be changed with the *kenelTimeSlice* function). VxWorks manual offers the following representative picture:



VxWorks provides the following functions related with task scheduling:

taskPriorityGet: Examine the priority of a Task.
sched_setparam: Set a task priority
shed_getparam: Get the scheduling parameters for an specific task.
sched_getscheduler: Get the current scheduling policy.
sched_get_priority_max: Get the maximum priority.
sched_get_priority_min: Get the minimum priority.
KenelTimeSlice: Change time slice for round-robin.
sched_rr_get_interval: If round-robin scheduling, get the time slice length.

6.- Thread Data Structures

As exposed before, µITRON and eCos don't provide specifically structures called task control blocks, but several information structures that can be gathered with different command requests. However, VxWorks provides a unified TCB structure, which can be accessed through a pointer obtained with a specific system call. Following the most important data structures are shown. Note that most of the information that they contain has been explained during the rest of the document and used for its redaction.

µITRON only specifies general direction of what information the TCB should include, although it stand that it is indeed implementation dependent:

- A group of flags indicating the states of the task
- Task priority level
- A storage region for program counter, general-purpose registers and stack pointer used by the task
- Task start address

Ecos source structure provide a header file with the implementation of this concept. However, this structures are almost identical to the ones provided by the ***eCos*** files themselves. Due to the lack of space, only the eCos ones will be shown:

```
typedef struct
{
    cyg_handle_t        handle;
    cyg_uint16          id;
    cyg_uint32          state;
    char                *name;
    cyg_priority_t      set_pri;
    cyg_priority_t      cur_pri;
    cyg_addrword_t      stack_base;
    cyg_uint32          stack_size;
    cyg_uint32          stack_used;
} cyg_thread_info;
```

```
    Cyg_Thread (
        cyg_thread_entry        *entry,         // entry point function
        CYG_ADDRWORD    entry_data,     // entry data
        cyg_ucount32            stack_size = 0, // stack size, 0 = default
        CYG_ADDRESS             stack_base = 0  // stack base  );

    Cyg_Thread (
        CYG_ADDRWORD        sched_info,     // Sched parms
        cyg_thread_entry            *entry,         // entry point function
        CYG_ADDRWORD        entry_data,     // entry data
        char                        *name,          // thread name
        CYG_ADDRESS             stack_base = 0, // stack base
cyg_ucount32            stack_size = 0  // stack size, 0 = default
        );
```

```
enum {                  // Thread state values

        RUNNING    = 0,         // Thread is runnable or running
        SLEEPING   = 1,         // Thread is waiting for something to
happen
        COUNTSLEEP = 2,         // Sleep in counted manner
        SUSPENDED  = 4,         // Suspend count is non-zero
        CREATING   = 8,         // Thread is being created
        EXITED     = 16,        // Thread has exited

        // This is the set of bits that must be cleared by a generic
        // wake() or release().
        SLEEPSET   = (SLEEPING | COUNTSLEEP)
    };
```

```
enum cyg_reason                 // sleep/wakeup reason codes
    {
        NONE,                   // No recorded reason
        WAIT,                   // Wait with no timeout
        DELAY,                  // Simple time delay
        TIMEOUT,                // Wait with timeout/tim expired
        BREAK,                  // forced break out of sleep
        DESTRUCT,               // wait object destroyed[note]
        EXIT,                   // forced termination
        DONE                    // Wait/delay complete
    };
```

Note again that the Hardware registers are included in a separate file with HAL details.

About ***VxWorks*** structures, only shown the TCB which holds main information, including hardware dependent which is not shown:

```
typedef struct windTcb
    {
    Q_NODE    qNode;                        /* 0x00: multiway q node: rdy/pend
q */
    Q_NODE     tickNode;          /* 0x10: multiway q node: tick q */
    Q_NODE    activeNode;         /* 0x20: multiway q node: active q */

    OBJ_CORE    objCore;          /* 0x30: object management */
    char *            name;                         /* 0x34: pointer to task name */
    int               options;          /* 0x38: task option bits */
    UINT             status;                       /* 0x3c: status of task */
    UINT             priority;          /* 0x40: task's current priority */
    UINT            priNormal;         /* 0x44: task's normal priority */
    UINT            priMutexCnt;     /* 0x48: nested priority mutex owned */
    struct semaphore * pPriMutex;               /* 0x4c: pointer to inheritance
    UINT           lockCnt;           /* 0x50: preemption lock count */
    UINT           tslice;                /* 0x54: current count of time slice */

    UINT16        swapInMask;      /* 0x58: task's switch in hooks */
    UINT16        swapOutMask;   /* 0x5a: task's switch out hooks */

    Q_HEAD *    pPendQ;                       /* 0x5c: q head pended on (if any)

    UINT             safeCnt;          /* 0x60: safe-from-delete count */
    Q_HEAD       safetyQHead;    /* 0x64: safe-from-delete q head */

    FUNCPTR      entry;  /* 0x74: entry point of task */

    char *     pStackBase;   /* 0x78: points to bottom of stack */
    char *     pStackLimit;  /* 0x7c: points to stack limit */
    char *     pStackEnd;     /* 0x80: points to init stack limit */

    int         errorStatus;    /* 0x84: most recent task error */
    int         exitCode;  /* 0x88: error passed to exit () */

    struct sigtcb *         pSignalInfo;              /* 0x8c: ptr to signal info for task */
    SEMAPHORE       selectSem;/* 0x90: select semaphore */
    struct selWkNode *    pSelWakeupNode;  /* 0xac: task's select info */

    UINT    taskTicks; /* 0xb0: total number of ticks */
    UINT    taskIncTicks;             /* 0xb4: number of ticks in slice */

    struct taskVar *      pTaskVar; /* 0xb8: ptr to task variable list */
    struct rpcModList *     pRPCModList;      /* 0xbc: ptr to rpc module statics */
    struct fpContext *   pFpContext;              /* 0xc0: fpoint coprocessor context
    struct __sFILE *     taskStdFp[3];
    int         taskStd[3];/* 0xd0: stdin,stdout,stderr fds */
    char **     ppEnviron;    /* 0xdc: environment var table */
    int           envTblSize;     /* 0xe0: number of slots in table */
    int           nEnvVarEntries; /* 0xe4: num env vars used */
    struct sm_obj_tcb *     pSmObjTcb;          /* 0xe8: shared mem
    int           windxLock;              /* 0xec: lock for windX */
    int           pad1[2];  /* 0xf0: padding to replace DBG_INFO */
    void *       pComLocal;            /* 0xf8: ptr to COM task-local
    REG_SET *     pExcRegSet;    /* 0xfc: exception regSet ptr
    int         reserved1; /* 0x100: possible user extension */
    int         reserved2; /* 0x104: possible user extension */
    int         spare1;                   /* 0x108: possible user extension
    int         spare2;                   /* 0x10c: possible user extension
    int         spare3;                   /* 0x110: possible user extension
    int         spare4;                   /* 0x114: possible user extension

    /* ARCHITECTURE DEPENDENT INFO FOLLOWING*/

typedef struct   /* TASK_DESC - information structure */
    {
    int         td_id;        /* task id */
    char *    td_name;   /* name of task */
    int         td_priority;  /* task priority */
    int         td_status;  /* task status */
    int         td_options; /* task option bits (see below) */
    FUNCPTR  td_entry; /* original entry point of task */
    char *    td_sp;        /* saved stack pointer */
    char *    td_pStackBase; /* the bottom of the stack */
    char *    td_pStackLimit;  /* the effective end of the stack */
    char *    td_pStackEnd;  /* the actual end of the stack */
    int         td_stackSize; /* size of stack in bytes */
    int         td_stackCurrent;/* current stack usage in bytes */
    int         td_stackHigh; /* maximum stack usage in bytes */
    int         td_stackMargin; /* current stack margin in bytes */
    int         td_errorStatus;  /* most recent task error status */
    int         td_delay;   /* delay/timeout ticks */
    } TASK_DESC;
```