NICHETASK

MULTITASKING SCHEDULER
TECHNICAL REFERENCE

interniche
technologies, inc.

51 E Campbell Ave
Suite 160
Campbell, CA. 95008

InterNiche Technologies Inc.
email: support@iniche.com
http://www.iniche.com
support: 408.540.1160
fax: 408.540.1161

Rev-10.2005

Portions of the InterNiche source code are provided under the copyright of the respective owners and are also acknowledged in the appropriate source files:

Trademarks

TABLE OF CONTENTS

1. OVERVIEW

This document describes NicheTask, the InterNiche multitasking scheduler software, and the interface specifications for it. This multitasking system contains only task control logic - no semaphores, mailboxes, etc. Tasks are scheduled in a round-robin manner and are not preempted. Once it gains control, each task runs until it voluntarily blocks. It is up to the programmer who uses this package to ensure that tasks do not run for indefinite amounts of time by relinquishing the CPU via calls to **tk_block()** or **tk_next()**, or one of the blocking **TK_** macros described below.

The API to the InterNiche tasking system is designed so that it can be easily mapped to a more sophisticated (i.e. real time) system by #defining task calls to the RTOS calls. All InterNiche applications which use Multitasking or support a Multitasking version are implemented using macros which start with the characters **TK_**, and ports are provided which show how these macros can be easily mapped to a variety of commercial RTOS systems. The **TK_** macros are defined in the section starting on page 7

This allows projects to begin development on an InterNiche system and “graduate” to a RTOS if it turns out to be required. Code written for the InterNiche system can usually be ported by creating the definitions and recompiling. All InterNiche networking code and applications are written using the **TK_** macros to facilitate such porting.

Tasks can be created and deleted dynamically by calls to the tasking API. Each task has a stack and a task control structure. Tasks are created via the routine **tk_new()**, which returns is a pointer to this structure. This pointer is thereafter used as a **task ID**.

The bulk of the tasking system is written in portable ANSI C code. Three low-level routines need to be provided if you port this system to an unsupported system. Since these routines are generally implemented in assembly language, this means you may need to hand edit these routines if you use an assembler not supported by InterNiche.

1.1 Source Files

The entire C version is implemented in two C language files. These are:

task.c
task.h

These are generally found in the InterNiche **\misclib** directory.

1.2 task.h Data Types

To call the tasking package from within a source file, you should include the file **\misclib\task.h** which defines the structures and types used in the tasking package. In InterNiche networking code this files is usually included in the scope of **ipport.h** or **osport.h**. There is a task control structure associated with each task instance, and a defined type **task**, which is a synonym of that structure. Each task structure contains a pointer to the tasks stack space (described below). This task structure and its stack area the are allocated as separate blocks via the **TK_ALLOC()** and **STK_ALLOC()** macros. Generally these are #defined to **calloc()** and **free()**.

1.3 Task Stacks

Each task structure contains a pointer to the task's stack. The pointer is a defined type **stack_t**, usually typedef'fed to an integer; thus each task's stack is an array of integers in memory. Note that some CPU/compiler combinations use stack space starting with the lowest address and go up, i.e. a **push** increments the stack pointer, and other systems start at the highest address of stack space and move down. The former are referred to as "bottom up" stacks and the later as "top down". You will need to indicate one of these options by #defining either **STK_TOPDOWN** or **STK_BOTTOMUP**.

1.4 The Task Control Structure

```
struct      task    {
    stack   *tk_fp;           /* task's current frame ptr */
    char    *tk_name;         /* the task's name */
    int     ev_flags;         /* flag set if task is scheduled */
    struct  task *tk_next;    /* pointer to next task */
    unsigned tk_count;        /* number of wakeups */
    unsigned *tk_guard;       /* pointer to lowest guard word */
    unsigned tk_size;         /* stack size */
    stack   tk_stack[1];      /* top of task's stack */
    };
```

A pointer to a task control structure is used as a process ID in this system. Task control structures form a circular list, chained by the **tk_next** member of the structure. The scheduler is a round robin scheduler; it simply loops through the circular list of tasks until it finds one which is runnable (that is, a task whose **ev_flg** is TRUE) and then switches to that task. A context switch merely consists of saving a small amount of state on the stack and calling the routine **tk_switch()** (which is called by **tk_block()** more about which you'll read later). When the task later runs again, it is by that call to **tk_switch()** returning.

When a task stack is allocated, it is filled with guardwords - a predefined constant used to track stack usage. The **tk_guard** field points to the last (lowest on **STK_TOPDOWN** systems) word on the stack. On every context switch, this word is checked to verify that it still contains a guardword. If it doesn't, the tasking package assumes that the task had a stack overflow and aborts the system with a call to the **panic()** routine. Guardwords are also used by the **tk_stats()** function to determine how much of the stack has been used.

Finally, there is a global variable, **tk_cur**, which is a pointer to the task control structure of the task which is currently running. For portability, it is recommended that you should only access **tk_cur** via macros and not **read** or **set** its members directly.

1.5 General Task Behavior

New tasks are created in a runnable state, and are not expected to return. Tasks which have finished their job and wish to self-destruct should do so by calling **tk_exit()**. Generally, task wakeups should be treated only as hints. When a task runs, it should try to discover why it was woken up and do the right thing. This includes being able to cope with a seemingly reasonless wakeup. Since tasks are forked as runnable, this is the suggested logic for a task routine:

```
task_routine(
{
/* declare local variables; */
…

    /* initialize task resources */

    while(1)
    {
            /* see if there’s work to do */
            if(work_to_be_done)
            {
                    /* do ongoing work */
            }
            tk_yield();       /* let rest of system run */
    }
}
```

There is a global variable called **TDEBUG** which is normally set to FALSE. When it is TRUE, every time a task runs or blocks, a message is printed on the display saying what it did and what the task’s name was. Of course, this makes it hard to see anything else which might be going on, but it can be useful to see what task is crashing the program, or what task runs when, and things of like nature.

1.6 Interrupts and Tasks

Interrupt handlers (ISRs) should never call any of the tasking functions other than **tk_wake()**. In general the only interaction between ISRs and tasks should be the ISR setting or clearing a task’s “runnable” flag via **tk_wake()**.

2. TK_ MACRO DEFINITIONS

If a task using application will potentially be ported to other multitasking systems, then it is obviously desirable to not to tie its design or source code directly to NicheTask's (or any other's) multitasking API. All InterNiche network applications are written using a set of macros which can be mapped to virtually any OS or RTOS. The macros, referred to for obvious reasons as the **TK_** macros, are described in this section.

Many of the **TK_** macros map directly to NicheTask calls as well as calls from many other popular embedded RTOS packages. Implementations are available from InterNiche upon request. Contact your InterNiche distributor for an up to date list of supported RTOS systems.

In InterNiche's networking code, the **TK_** macros are generally defined in a file named **osport.h**. Projects which support multiple RTOS will have either multiple osport.h files available, or a single **osport.h** file with **#ifdefs** for the different systems. This **osport.h** file should be included in any file which uses the **TK_** macros. In InterNiche applications this is usually done by including **osport.h** in **ipport.h**; which is ultimately included in nearly all InterNiche source files.

There are two classes of **TK_** macros - those which map to a procedure call and those which are used to declare an object. The object declarations are used so that task structures, Ids, and entry points can be declared in a generic way and will not need to be redefined in the source code when recompiling with a new RTOS. The object declarations are listed here, along with their definition for NicheTask:

```
#define TK_ENTRY(name) int name(int parm)              /* function declaration */
#define TK_ENTRY_PTR(name) int(*name)(int) /* pointer to function */
#define TK_OBJECT(name) task * name                    /* task object */
#define TK_OBJECT_PTR(name) task ** name               /* pointer to object */
```

```
#define TK_OBJECT_REF          TK_OBJECT
         - or -
#define TL_OBJECT_REF          TK_OBJECT_PTR
```

Several InterNiche project directories use these macros to share a single "**main**" C files across several operating systems. This file, named **netmain.c**, declares an array of structures, each of which contains the required parameters to create one of the tasks needed in an embedded networking system. The **netmain.c** files also contains a **netmain()** routine invoked by system at startup. It uses a **for** loop to create a task for each entry defined in the array, it then calls XXXXX() in XXXXX.c to create any application server tasks that have been enabled through **ipport.h**. The individual task objects and task information structures are defined in the application directories. The keyboard i/o task in tk_misc.c provides an example of how this can be done. Since **netmain.c** is written using the **TK_** macros, it can be compiled and used with multiple operating systems.

2.1 Portable Task Description Structure

The structure for the table with an entry for each Internet task/thread is named **inet_taskinfo**. It is defined in **osport.h**, even though it is generally identical from port to port. This allows the porting engineer to add fields if needed. Since the structure array is initialized in **netmain.c** it should start with the same values in the same order in all ports.

The default for this is:

```
struct inet_taskinfo {
  TK_OBJECT_PTR(tk_ptr)        /* pointer to static task object */
  char * name;                 /* name of task */
  TK_ENTRY_PTR(entry);         /* pointer to code to start task at */
  int priority;                /* priority of task */
  int stacksize;               /* size (bytes) of task's stack */
};
```

All the members of this structure are declared with standard "C" types or **TK_** macros for portability. Not all fields are used on all tasking systems, for example the **priority** field is not used on round robin schedulers like NicheTask. Even in these cases, the unused fields should be left in place for portability.

2.2 TK_ Task Control Definitions

The **TK_** macros which map to procedure calls fall into two sub-classes - those which create & delete tasks, and those which transfer program control. In tasking systems which do not support dynamic task creation and deletion, the create macros may simply finish construction of task management objects or mark a task as runnable - they don't necessarily have to do the actual creation. The InterNiche networking software, including **netmain.c**, does not delete tasks, so the delete macros may be simply #defined away.

The procedural **TK_** macros are listed here and described in detail in the next section. This example is for the NicheTask system. Some aspects of these definitions are explained below.

```
int TK_NEWTASK(struct inet_taskinfo * nettask);

/* define TK_ macros to NichTask: */
#define TK_APP_BLOCK(event) tk_ev_block(event)
#define TK_APP_WAKE(event) tk_ev_wake(event)
#define TK_SLEEP(ticks) tk_sleep(ticks)
#define TK_NETRX_BLOCK() tk_ev_block(&rcvdq)
#define TK_CONS_BLOCK() tk_next()
#define TK_YIELD() { tk_wake(tk_cur); tk_block(); }
#define TK_WAKE() tk_wake()
#define TK_RETURN_ERROR() return (-1)              /* task error return */
#define TK_RETURN_OK() parm++; return (0)          /* task OK return */
```

In this case, **TK_NEWTASK()** is a function declaration and not a macro. This is primarily a matter of style - it could be done either way. Since this is only called a few times, and only at system initialization time, the readability and smaller code size of a routine was chosen in this port over the faster execution time of a macro.

Another bit of code worth explaining is the “parm++;” in **TK_RETURN_OK**. To understand this, review the example definition of **TK_ENTRY()**. It defines a routine with a single passed parameter of type **int**. Since the **TK_ENTRY** macro is used in **netmain.c** to define task’s entry points and the parameter is not used, a simple **return(0)** statement would result in many C compilers generating a warning about an unused parameter. The **netmain.c** tasks do not actually use the parameter since not all multitasking systems support a parameter. By performing an increment operation on the parameter in the return macro, the compiler warning is avoided.

2.3 Tasking System Designs - Spinning vs. Event-blocking

Before describing the **TK_** macros in detail, it is worth explaining the underlying assumptions about how they will be used. The main design goal of these macros is to support InterNiche networking code on a wide variety of systems without any code modifications other than the macro implementations themselves. As with any such system there are some simplicity vs. performance trade-offs which warrant explaining.

The simplest task model used by the InterNiche networking code is one where each task remains always ready to run. The tasks run for a reasonable period of time (as determined by the porting programmer) or, more often, until there is no more work to be done. The task then blocks, giving other tasks the opportunity to run in round-robin fashion. Each task has a chance to run in the same fashion.

The advantage of this approach is simplicity - there are no decisions to be made about how and when to suspend or wake up tasks. Each task essentially makes the decision for itself, blocking briefly when there is no work and running when there is. The disadvantage is the inefficiency of waking every task on every pass through the scheduler. This inefficiency is not as bad as it sounds. The InterNiche networking tasks are written to quickly determine if they should run or not, and return immediately if not. The C code when well optimized on a RISC processor can do the call-test-return sequence in as little as three CPU instructions. This means that this “infinite spinning” system is an excellent choice for most applications.

Some systems, however, require that the tasks not be runnable when there is no work to be done. One potential reason for this is a battery powered device which wishes to be able to shut down the CPU to save power, and only does so when all tasks are suspended. The InterNiche networking code and the **TK_** macros can work on such event based blocking systems if the underlying tasking package is capable of supporting tasks which block pending an event. NicheTask has this capability. Even RTOS packages which don’t directly support blocking on an event can often be supported by creating additional layers of code to do this.

The basic philosophy of InterNiche networking tasks is that each task services a queue or linked list of structures, where each list item represents some aspect of the network activity which may shortly require CPU cycles. The most common examples of these items are TCP connections with received data in the socket structures, and network packets received from hardware devices. When no such circumstance is pending, all tasks suspend themselves with the **TK_APP_BLOCK()** or the **TK_NETRX_BLOCK()** macros, and the system can be powered down until incoming network traffic (or a user command) wakes it up. In these cases the two **_BLOCK** macros are expected to both be mapped to the same tasking system blocking call - **tk_ev_sleep()** in the case of NicheTask.

2.4 Priorities

In tasking environments which support multiple priorities, the best philosophy is to create all the network tasks at the same priority level and let them run round-robin style. Setting them to different priorities creates the potential for a high priority task that ends up with a large amount of input to lock out lower priority tasks and thus hurt system performance. Any application needs the application's task, the network task, the timer task, and the interrupt system to all get reasonable amounts of CPU time or the overall performance of the application will suffer.

It is also usually not required to run the network tasks at a high priority in a prioritized system. The TCP/IP Networking protocols and their applications are designed to allow for wide variances in event timing. Any event (such as a received packet) which is delayed or dropped will be handled by the protocols. It is generally fine to save the high priority tasks for whatever applications make up the systems primary function, and let the network catch up in the systems spare cycles.

3. THE TK MACROS

This section contains detailed descriptions of the **TK_** macros. The syntax illustrated in the synopsis section assumed the macro is treated like a function.

NAME

TK_NEWTASK()

SYNTAX

int TK_NEWTASK(struct inet_taskinfo * nettask);

DESCRIPTION

TK_NEWTASK() is called to create the actual task entity. This may include allocating a task structure, a stack, or other resources for the task, and setting it up to run. Note that in some tasking systems the task structures and stack memory are statically declared via **TK_OBJECT()** and only need to be activated, while in others such as NicheTask the tasks are allocated from heap. In this later case, the task's identifier is an unassigned pointer to the task which is filled in by **TK_NEWTASK()** before it returns.

The passed pointer is to an **inet_taskinfo** structure (described in the pervious section) which contains information used to set up the task. Not all the information in this structure is used by every port.

Tasks should be created ready to run, and may be started by the system at any time after creation. Tasks should be coded to test for any required resources or conditions as they start executing. An example of this is the **netmain.c** application tasks, which test the global variable **NET_READY** before commencing network IO.

RETURNS

TK_NEWTASK should return a zero if the task was successfully created, and a negative one (-1) if not. Specific error codes or task Ids which need to be returned can be saved in port-specific fields added to the end of the passed structure.

NAME

TK_APP_BLOCK()

SYNTAX

void TK_APP_BLOCK(void * event)

DESCRIPTION

This macro is called by an application when it has no more immediate work to do. The passed **event** parameter is a pointer to a data object which may optionally be used to wake the task when some event requiring the tasks attention occurs.

The porting engineer needs to decide before implementing this macro if the tasking system will adhere to one of the tasking models described in the section, Tasking System Designs - Spinning vs. Event-blocking on page 9, or do something else entirely.

In a "spinning" task system design, this macro may opt to ignore the passed flag and simply relinquish the CPU to other tasks. In systems where different tasks have different priorities, care should be taken to ensure that lower priority tasks get an opportunity to run. This could mean actually putting the task to sleep for a brief interval.

In event based blocking systems this macro should record the passed pointer and block the calling task until a call is made to **TK_APP_WAKE()** with the same **event** pointer.

NAME

TK_APP_WAKE()

SYNTAX

void TK_APP_WAKE(void * event)

DESCRIPTION

This is called to awaken a task which has been blocked by a previous event blocking call, such as **TK_APP_BLOCK()** or **TK_NETRX_BLOCK()**. Calling this is only required on systems which use event blocking. Systems which use the infinitely spinning approach do not strictly need to call this, although it is a good idea for portability reasons.

The address passed should be the same address that was passed to the **_BLOCK** call. All tasks which have block on this address are set as runnable. On Some RTOS systems a higher priority task which is asleep on the event passed to **TK_APP_WAKE()** may run before **TK_APP_WAKE()** returns, so code should be designed to allow this.

NAME

TK_SLEEP()

SYNTAX

void TK_SLEEP(long ticks)

DESCRIPTION

Calling this causes the calling task to suspend for the specified number of system clock tick. On InterNiche networking systems clocks ticks are tracked by the variable **cticks**, and the frequency is defined by **TPS** (ticks per second).

Tasks put to sleep with this call may be awakened before the indicated time by a call to **TK_WAKE()**. They are not awakened by calls to **TK_APP_WAKE()**.

NAME

TK_NETRX_BLOCK()

SYNTAX

void TK_NETRX_BLOCK(void * event)

DESCRIPTION

The call is semantically identical to **TK_APP_BLOCK()**. It is only called from the InterNiche TCP/IP stack's received packet handling task and should not be used by any other code.

It is a separate macro because on some systems the received packet handler may be implemented as a special case, getting a higher priority or faster wake-ups than the applications which call **TK_APP_BLOCK()**. On systems with polled network devices (for example ring-buffer Ethernet chips) this routine is a good place to do the device polling.

The event for this call is always the address of the received packet queue (**rcvdq**).

NAME

TK_CONS_BLOCK()

SYNTAX

void TK_CONS_BLOCK(void)

DESCRIPTION

This is another special-case blocking routine. In this case, it blocks waiting for character input from the system console. When a character is ready to read, it should return. The calling code should still verify that a character is ready before calling **getch()** (or whatever console ready mechanism it uses) to avoid blocking inside the **getch()** function.

Blocking inside a Keyboard Reading routine is one of the most common mistakes in implementing embedded multitasking packages.

On systems where the console input is marked by asynchronous events (e.g. interrupts), this macro may be mapped to an event blocking routine. Other systems may need to poll a keyboard flag or buffer state to determine when it is time to return to the calling task.

In general as with **TK_NETRX_BLOCK(),** this routine should not be called by tasks other than the one it is tailored to, in this case the InterNiche port console handler.

NAME

TK_OBJECT_REF

SYNTAX

#define TK_OBJECT_REF   TK_OBJECT

OR

#define TK_OBJECT_REF   TK_OBJECT_PTR

DESCRIPTION

This is a marco which is an indirection to either the TK_OBJECT or TK_OBJECT_PTR macro's that are used to declare an object. TK_OBJECT_REF declares the task identifies in the tk_wait_event structure which is used to implement a version of tcp_sleep() and tcp_wakeup(), which is portable across all RTOSs. Some RTOSs like PSOS identify tasks with a long, or with an integer like in the case of uC-OS, however some other RTOSs like ThreadX identify tasks with a pointer to a structure.

Hence this macro needs to setup to be either a TK_OBJECT or TK_OBJECT_PTR whichever is the principal identifying method for the target RTOS.

NAME

TK_THIS()

SYNTAX

TK_OBJECT_REF   TK_THIS (void)

DESCRIPTION

This call is used in the tcp_sleep() and tcp_wakeup() implementation which is portable across various RTOS'es. TK_THIS() is expected to identify a task in terms of the TK_OBJECT_REF which is a generic identifier for the task as described earlier. On some RTOS'es this macro maps directly while some others might need a wrapper around the call.

NAME

TK_WAKE()

SYNTAX

void TK_WAKE(TK_OBJECT_PTR Id)

DESCRIPTION

This is called to awaken a task which has been blocked by a previous event blocking call, such as **TK_BLOCK()** or **TK_NETRX_BLOCK()**. Calling this is only required on systems which use event blocking. Systems which use the infinitely spinning approach do not strictly need to call this, although it is a good idea for portability reasons.

The address passed should be the same address that was passed to the **_BLOCK** call. All tasks which have block on this address are set as runnable. On Some RTOS systems a higher priority task which is asleep on the event passed to **TK_WAKE()** may run before **TK_WAKE()** returns, so code should be designed to allow this.

Note that this calls strictly expects a pointer to a task object as its argument.

NAME

TK_WAKE_EVENT ()

SYNTAX

void TK_WAKE_EVENT(TK_OBJECT_REF event)

DESCRIPTION

This is called to awaken a task which has been blocked by a previous event blocking call, such as **TK_BLOCK()** or **TK_NETRX_BLOCK()**. Calling this is only required on systems which use event blocking. Systems which use the infinitely spinning approach do not strictly need to call this, although it is a good idea for portability reasons.

The object reference passed should be the same object reference that was passed to the **_BLOCK** call. All tasks which have block on this object reference are set as runnable. On Some RTOS systems a higher priority task which is asleep on the event passed to **TK_WAKE_EVENT ()** may run before **TK_WAKE_EVENT ()** returns, so code should be designed to allow this.

Note that this call strictly expect the argument to be of type TK_OBJECT_REF (which is again an indirection to TK_OBJECT or TK_OBJECT_PTR depending on the target RTOS).

NAME

TK_YIELD()
tk_yield()

SYNTAX

void TK_YIELD(void)

DESCRIPTION

TK_YIELD() is called when the task code wants to wait for something to occur - a situation often referred to as a “busy wait”. The **TK_YIELD()** primitive must give other tasks a chance to run, yet resume the calling task in a short interval. On a round-robin system like NicheTask this is easy - you simply mark to current task as runnable an call the round-robin scheduled.

On an RTOS where tasks have priorities, this can be somewhat trickier to implement. These systems sometimes support a call which will let tasks of equal or greater priority run, by not lower priority tasks. A task spinning on such a **TK_YIELD()** macro would never allow a lower priority task to run.

One remedy for this is to code the **TK_YIELD()** macro to put the task to sleep for a single clock tick. This will force it to wait a reasonable interval during which lower priority tasks may potentially get some cycles. The draw back is that even when the system has nothing else to do, the task spinning on will never be able to utilize all the CPUs power - it will always spend a certain amount of time gratuitously blocked.

The **tk_yield()**macro (same name in lower case) is identical to the uppercase version. It is supported for historical reasons.

NAME

TK_RETURN_ERROR()
TK_RETURN_OK()

SYNTAX

```
void TK_RETURN_ERROR(void);
void TK_RETURN_OK(void);
```

DESCRIPTION

These macros are used in place of a **return** statement in task routines declared with **TK_ENTRY()**. In most tasking systems a task should never return, however the **return** statement is often required to avoid compiler warnings. Both error and non error varieties are provided for completeness.

4. USER TASKING FUNCTIONS

Descriptions of functions in the tasking system follow.

NAME

tk_new()

SYNTAX

```
task *tk_new(task *prev_task,
     int (*entry_point)(),
     unsigned stack_size,
     char *name,
     unsigned arg);
```

DESCRIPTION

This call creates a new task, setting up a task control structure and a stack for it. It inserts it in the linked list of tasks directly after the task specified by **prev_task**. **entry_point** is the pointer to the routine which implements this task. **arg** is the argument passed to **entry_point**.

tk_new() builds a stack frame for this function by calling the lower level function **tk_frame()**, so that the first time the task runs it can utilize its stack. It can store its state on the stack, have local variables, and in general, do most things that any normally called C routine could do, except that it must never return.

The **stack_size** parameter indicates the number of bytes of stack which **tk_new()** should allocate. This stack is used by the tasks local variables, by functions called by the task, and as the interrupt stack for any interrupts which occur while that task is running. 800 to 1000 bytes seem to be good minimum sizes.

This is an example of **entry_point** for a task which will print a message and then loop forever, printing further messages. Tasks are set to runnable when created, so it does not need to be awakened after it is created with **tk_new()**.

```
task1_entry_point()
{

     printf("Hello world. \n");
     printf("task1: starting up. \n");

     while(1)
     {
         printf("… still running…");
         tk_yield()'
     }
}
```

It is generally good form to enclose the main body of the task in a **while** or **for** loop. Although tasks should never return, some compilers may also require a return statement at the end of the function to suppress compiler warnings. Ironically, other compilers will warn that the return statement, if present, is unreachable. The InterNiche **TK_** macros define **TK_RETURN** macros, so these compiler dependencies can be dealt with in a single include file rather than in every task's source.

Finally, the **name** parameter is a string which has a textual name for the task and is sometimes useful for debugging.

NAME

tk_init()

SYNTAX

task *tk_init(unsigned stack_size);

DESCRIPTION

This routine initializes the tasking system. It builds a task control structure for the "currently running" task, gives it a name, **main**, and leaves it running on the system stack. It is important to realize that the stack in use when **tk_init()** is called becomes the stack of the created task - the task named **main** - and thus should not be returned to heap or otherwise used later for any other purpose.

The purpose of this stack reuse feature is to conserve space on low-memory systems. The compiler and/or bootstrap code often allocate a reasonably good sized stack, and there is usually no reason why this stack cannot be used by a task during runtime. In cases where the boot-up stack is not appropriate for task use, the OS port code should install a stack which is suitable prior to calling **tk_init()**.

The **stack_size** parameter should be the number of bytes available on the stack when **tk_init()** is called. This function should be called before any other routine in the tasking system.

RETURNS

If successful, it returns a pointer to the task control structure that it built. If it fails (for example it cannot allocate the task structure) it returns NULL.

NAME

tk_ev_block()

SYNTAX

```
void tk_ev_block(void * event);
```

DESCRIPTION

The routine causes the calling task to suspend until another task calls **tk_ev_wake()** with an identical **event** value. The **event** value is usually a pointer to a buffer or structure which is controlled by the calling task. The task's **wake** flag is cleared, and any previous event the task was sleeping on is cleared.

NAME

tk_ev_wake()

SYNTAX

void tk_ev_wake(void * event);

DESCRIPTION

This routine wakes any tasks which have been blocked by a previous call **tk_ev_block()** with the same event value. All tasks blocked on the event are awakened. The event is cleared in the awakened task's structure, so future calls to **tk_ev_wake()** will NOT awaken the task again unless they have made another call to **tk_ev_block()**.

NAME

tk_block()

SYNTAX

```
void tk_block(void);
```

DESCRIPTION

tk_block() takes no arguments and returns no value. It blocks the currently running task. Since the tasking system is non-preemptive, this is the only way for another task to gain control of the processor. Other tasking system entry points which swap tasks do this by first setting a wake condition and then calling **tk_block()**. This routine returns the next time the task which blocked runs.

tk_block() contains the heart of the scheduler. It basically runs through the circular list of tasks until it finds a runnable task and then does a context switch to that task, which then sees its last call to **tk_block()** return. **tk_block()** calls **tk_switch()** to perform the actual context switch.

NAME

tk_exit()

SYNTAX

```
void tk_exit(void);
```

DESCRIPTION

This function causes the current task to die. When another task becomes runnable, this task's task control structure and stack will be deallocated. This routine should not be called from interrupt level and no further references should be made to this task's task control structure after this call is made.

tk_exit() never returns.

NAME

tk_wake()

SYNTAX

void tk_wake(task * tk);

DESCRIPTION

This routine is a macro or higher level routine implemented on **tk_block()**. It merely sets the tasks **event flag** in the task control structure to which **tk** points to indicate that the task should run. The next time the scheduler runs, this task will be considered runnable.

This routine can be called from interrupt handlers, and is the preferred mechanism by which an ISR should initiate system processing. An example of a usage of this routine is by a mac driver interrupt handler, which, when a good packet is received, enqueues the packet on the received packet queue and wakes the task which handles received packets.

NAME

tk_next()

SYNTAX

void tk_next(void);

DESCRIPTION

tk_next() performs a **tk_wake(tk_cur)** and a **tk_block()**. It essentially yields the processor to let other tasks run, but wakes up the current task before yielding so that it regains control of the processor after an unspecified amount of time.

Functionally identical to **tk_yield()**, except that it is always defined as a routine (not a macro) for portability reasons. The **TK_** macro **TK_YIELD()** may be implemented on NicheTask by #defining it to **tk_next()**, or defined as a macro which directly calls the **tk_wake()** and **tk_block()** functions.

NAME

tk_kill()

SYNTAX

void tk_kill(task * tk);

DESCRIPTION

This function kills a task. The task is immediately removed from the list of tasks and its stack is deallocated. Tasks should not call **tk_kill** with their own task pointer, they should use **tk_exit()** instead.

NAME

tk_sleep()

SYNTAX

```
void tk_sleep(unsigned long ticks);
```

DESCRIPTION

This function puts a task to sleep for the specified interval, given by the number of clock ticks passed. The number of these ticks per second is given by the constant **TPS**, so if you want to suspend a task for three seconds you could code the **tick** parameter as (3*TPS). The code which determines the exact duration that the task will sleep has an accuracy of +/- one (1) tick. Of course since the system is on-preemptive, another task “hogging” the CPU could cause the task to sleep indefinitely.

NAME

tk_stats()

SYNTAX

int tk_stats(void * io_dev);

DESCRIPTION

This macro prints statistics about all tasks in the list. It prints the name of each task, the allocated stack size and the number of bytes of stack which have actually been used by the task. The **io_dev** parameter is a system dependent descriptor for an output device to use, usually an InterNiche "generic" IO device as implemented in **\misclib\in_util.h**. If the **io_dev** pointer is NULL, the system should print the statistics to the default output device (if any).

5. LOW-LEVEL ROUTINES

The following functions are internal to the tasking system only and should not be called from outside of it. These routines are generally not portable across processors (or sometimes even assemblers), and as such need to be implemented for each target system. These are usually written in machine (assembly) language.

NAME

tk_frame()

SYNTAX

tk_frame(task *tk, int (*entry)(), int arg);

DESCRIPTION

This routine builds the actual stack frame for a task. It builds the frame for the task whose control structure is pointed to by **tk**. **entry** is the entry point of the task. **arg** is the value passed to the routine when it runs for the first time. The task is not set to runnable by **tk_frame()**; this must be done by the caller if desired.

This is called by **tk_new()**.

NAME

tk_switch()

SYNTAX

tk_switch(task *tk);

DESCRIPTION

This routine performs the actual context switch to the task whose control block is pointed to by **tk**. When the current task runs again, the call that was made to **tk_switch()** will return.

This is called from inside **tk_block()**.

NAME

tk_getsp()

SYNTAX

stack_t * tk_getsp(void);

DESCRIPTION

This returns the current stack pointer. It should just place the current stack pointer into a return register as expected by the C compiler. The value of the system's stack may change when this call returns, but the calling code allows for this.

This is called from **tk_init()** to find the main task's stack.

Index