uMT Reference Manual

This documents is the reference manual for the uMT callable primitives (implemented as C++ methods). This is a preliminary version which might not be including all the calls: please refer to ***uMT.h*** for the full list.

A Getting Started document is available as well.

uMT is a young software and, as consequence, no extensive use (and debugging) has been done yet. As a consequence users must be aware of this and they are suggested to contact the Author in case of bugs or issues.

Please also note that this an EDUCATIONAL tool, not designed for industrial or state-of-the-art application (nor life or mission critical application!!!) and not fully optimized.

uMT v2.5.0 – Doc v2.5.0 – 07 June 2017

uMT

Micro Multi Tasker (uMT) is a preemptive, timesharing, soft real-time (not deterministic) multi tasker specifically designed for the Arduino environment. uMT currently works with the AVR microprocessor family (***Arduino Uno*** and ***Arduino Mega2560***) and for the SAM/SAMD architecture (***Arduino Due*** *and* ***Zero***).

Please note that the SAMD porting has not been tested! (the Author does not own a Zero board).

uMT is a simplified rewriting of a 30 years old multi tasker developed by the Author in his youth (!) which was originally designed for the Intel 8086 architecture (and then ported to protected mode 80386, M68000, MIPS R3). uMT is significantly simpler than the original and it has been designed for ease of use and power for the small environment of Arduino microprocessors and boards. Last, uMT is written in a simple C++ versus its C language ancestor to hide complexity and increase ease of use.

Table of Contents

RETURNED ERROR CODE

All uMT kernel primitives return an error code. In the table below the full list of error codes is listed.

As a general statement, the kernel returns E_SUCCESS for success. E_NOT_INITED is also returned if the Kn_Start() call has not been performed yet (and this error code is not listed in each individual call).

```
enum Errno_t
{
/* 00 */ E_SUCCESS,                    // SUCCESS
/* 01 */ E_ALREADY_INITED,             // KERNEL already inited
/* 02 */ E_ALREADY_STARTED,            // TASK already started
/* 03 */ E_NOT_INITED,                 // KERNEL not inited
/* 04 */ E_WOULD_BLOCK,                // Operation would block calling TASK
/* 05 */ E_NOMORE_TASKS,               // No more TASK entries available
/* 06 */ E_INVALID_TASKID,             // Invalid TASK Id
/* 07 */ E_INVALID_TIMERID,            // Invalid TIMER Id
/* 08 */ E_INVALID_MAX_TASK_NUM,       // Not enough TASK entries configured in the Kernel
(Kn_Start) or too many
/* 09 */ E_INVALID_SEMID,              // Invalid SEMAPHORE Id
/* 10 */ E_INVALID_TIMEOUT,            // Invalid timeout (zero or too large)
/* 11 */ E_OVERFLOW_SEM,               // Semaphore counter overflow
/* 12 */ E_NOMORE_TIMERS,              // No more Timers available
/* 13 */ E_NOT_OWNED_TIMER,            // TIMER is not owned by this task
/* 14 */ E_TASK_NOT_STARTED,           // TASK not started, cannot ReStartTask()
/* 15 */ E_TIMEOUT,                    // Timeout
/* 16 */ E_NOT_ALLOWED,                // Not allowed [Tk_ReStartTask() suicide]
/* 17 */ E_INVALID_OPTION,             // Invalid additional option
/* 18 */ E_NO_MORE_MEMORY,             // No more memory available [Tk_CreateTask()]
/* 19 */ E_INVALID_STACK_SIZE,         // Invalid STACK size [Tk_CreateTask()]
/* 20 */ E_INVALID_MAX_TIMER_NUM,      // Invalid max Timer number [Kn_start()]
/* 21 */ E_INVALID_MAX_SEM_NUM         // Invalid max Semaphore number [Kn_start()]
};
```

KERNEL MANAGEMENT

Kernel management provides the following primitives:

NAME	DESCRIPTION
Kn_Start()	Start uMT Kernel
Kn_Inited()	Return if uMT has been initialized
Kn_PrintInternals()	Print uMT internal kernel tables on Serial, return E_SUCCESS
Kn_GetConfiguration()	Return uMT kernel configuration
Kn_PrintConfiguration()	Print uMT kernel configuration on Serial, return E_SUCCESS
isr_Kn_GetVersion()	Return uMT version, a 16 bit value organized by x.y.z (8.4.4 bits).
isr_Kn_FatalError()	Print an optional message to Serial, a fatal error message and reboot
isr_Kn_Reboot()	Reboot (restart) the system
isr_Kn_IntLock()	Disable global interrupts and return previous interrupt mask
isr_Kn_IntUnlock()	Restore previous interrupt mask (possibly enabling interrupts)
isr_Kn_GetKernelTick()	Return internal kernel counter (usually in milliseconds)

NAME

Kn_Start()

SYNTAX

```
Errno_t       Kn_Start();
Errno_t       Kn_Start(Bool_t _TimeSharingEnabled);
Errno_t       Kn_Start(uMTcfg &Cfg);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
TimeSharingEnabled	IN	TRUE to enable, FALSE to disable
Cfg	IN	Configuration setting

DESCRIPTION

Initialize the uMT subsystem.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_INVALID_MAX_TASK_NUM	Task number too big
E_INVALID_MAX_TIMER_NUM	Timer number too big
E_INVALID_MAX_SEM_NUM	Semaphore number too big
E_NO_MORE_MEMORY	Not enough memory for Kernel structures

EXAMPLE

NAME

Kn_Inited()
Kn_PrintInternals()
Kn_GetConfiguration()
Kn_PrintConfiguration()
isr_Kn_GetVersion()
isr_Kn_FatalError()
isr_Kn_Reboot()
isr_Kn_IntLock()
isr_Kn_IntUnlock()
isr_Kn_GetKernelTick()

SYNTAX

```
Bool_t              Kn_Inited();
Errno_t             Kn_PrintInternals()
Errno_t             Kn_GetConfiguration()
Errno_t             Kn_PrintInternals(Bool_t PrintMaxUsedStack = FALSE);
uint16_t            isr_Kn_GetVersion()
void                isr_Kn_FatalError()
void                isr_Kn_Reboot()
CpuStatusReg_t      isr_Kn_IntLock()
void                isr_Kn_IntUnlock(CpuStatusReg_t Flags)
Timer_t             isr_Kn_GetKernelTick()
```

DESCRIPTION

RETURNED VALUE	NAME	DESCRIPTION
Bool_t	Kn_Inited()	Return if uMT has been initialized
Errno_t	Kn_PrintInternals()	Print uMT internal kernel tables on Serial, return E_SUCCESS
Errno_t	Kn_GetConfiguration()	Return uMT kernel configuration
Errno_t	Kn_PrintInternals()	Print uMT kernel configuration on Serial, return E_SUCCESS. If “PrintMaxUsedStack” is set to TRUE, for each task entry a calculation of the used stack is performed. This heuristic calculation is achieved by setting, at task’s creation, the task’s stack area to a well-defined value and then scanning the area to discover which part has not been modified yet. The mechanism is not error proof but it gives anyway a good estimate.
uint16_t	isr_Kn_GetVersion()	Return uMT version, a 16 bit value organized by x.y.z (8.4.4 bits).
void	isr_Kn_FatalError()	Print an optional message to Serial, a fatal error message and reboot
void	isr_Kn_Reboot()	Reboot (restart) the system
CpuStatusReg_t	isr_Kn_IntLock()	Disable global interrupts and return previous interrupt mask
void	isr_Kn_IntUnlock()	Restore previous interrupt mask (possibly enabling interrupts)
Timer_t	isr_Kn_GetKernelTick()	Return internal kernel counter (usually in milliseconds)

TASK MANAGEMENT

Task management provides the following primitives:

NAME	DESCRIPTION
Tk_CreateTask()	Create a new task
Tk_DeleteTask()	Delete an existing task
Tk_StartTask()	Start a just created task
Tk_GetMyTid	Return my task id number
Tk_Yield()	Release the processor and perform a rescheduling (with a possible task round robin)
Tk_ReStartTask()	Restart a task
Tk_GetActiveTaskNo()	Return the total number of active tasks (excluding IDLE task)
Tk_SetTimeSharing()	Set the Timesharing flag
Tk_GetTimeSharing()	Return the Timesharing status flag
Tk_SetPreemption()	Set preemption flag
Tk_GetPreemption()	Return preemption status flag
Tk_SetBlinkingLED()	Set BlinkingLED flag
Tk_GetBlinkingLED()	Return BlinkingLED status flag
Tk_SetPriority()	Set task priority
Tk_GetPriority()	Return task priority
Tk_SetParam()	Set task launch parameter
Tk_GetParam()	Return own launch parameter
Tk_GetTaskInfo()	Return task's info
Tk_PrintInfo()	Print on Serial task's info

NAME

Tk_CreateTask()

SYNTAX

```
Errno_t Tk_CreateTask(
          FuncAddress_t StartAddress,
          TaskId_t &Tid,
          FuncAddress_t _BadExit = NULL,
          StackSize_t _StackSize = 0);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
StartAddress	IN	The start address (a C/C++ function pointer) of the new task.
Tid	OUT	The newly created task ID.
_BadExit	IN	The address (a C/C++ function pointer) of a function which will be called if the task main entry point incorrectly performs a "return". If missing, the uMT BadExit() routine will be called (rebooting the system).
StackSize	IN	The stack size in bytes (missing or 0 to use default value).

DESCRIPTION

Create a new task in a CREATED state. A subsequent call to Tk_StartTask() is required to put the task in the READY to run list.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_INVALID_STACK_SIZE	Invalid (too big) stack size
E_NOMORE_TASKS	Task table full, cannot create any new task
E_NO_MORE_MEMORY	Not enough memory for Kernel structures

EXAMPLE

NAME

Tk_DeleteTask()

SYNTAX

```
Errno_t Tk_DeleteTask(TaskId_t Tid);
Errno_t Tk_DeleteTask();
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
Tid	IN	The task ID to be deleted

DESCRIPTION

Delete a task. In the form without a parameter, delete the calling task.
Note that Arduino loop() task (Tid 1) cannot be deleted.
Task's stack area is released and the task slot is available for a new task creating.
Any pending timers is released and the deleted task is removed from any queue.

Any other resource owned by the task (including semaphores) are NOT released. As a consequence, a proper clean up logic must be implemented before/after calling ***Tk_DeleteTask***().

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_INVALID_TASKID	Invalid Task ID

EXAMPLE

NAME

Tk_StartTask()

SYNTAX

```
Errno_t       Tk_StartTask(TaskId_t Tid);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
Tid	IN	Task ID to start

DESCRIPTION

Start a task previously created.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_INVALID_TASKID	Invalid task ID
E_ALREADY_STARTED	Task already started

EXAMPLE

NAME

Tk_GetMyTid()

SYNTAX

```
Errno_t       Tk_GetMyTid(TaskId_t &Tid);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
Tid	OUT	Returned Task ID

DESCRIPTION

Return in Tid the Task ID of the caller.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_INVALID_TASKID	Invalid task ID
E_ALREADY_STARTED	Task already started

EXAMPLE

NAME

Tk_Yield()

SYNTAX

```
Errno_t       Tk_Yield();
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION

DESCRIPTION

Yield (relinquish) the processor in favor of another task. If this task is the only READY to run or it is the highest priority task, the control returns immediately. In any case, a task switch is performed. If there are other task READY to run, the caller task is inserted in the READY queue according to its priority (after the last task with same or higher priority).

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success

EXAMPLE

NAME

Tk_ReStartTask()

SYNTAX

```
Errno_t       Tk_ReStartTask(TaskId_t Tid);
Errno_t       Tk_ReStartTask();
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
Tid	IN	Task ID to re-start

DESCRIPTION

Restart the task specified by Tid or the caller task if invoked without parameters.
A task restart operation will reset the initial stack pointer to the entry routine of the task but it will not change current task priority. It remains in the application responsibility to clean up any other application related resource (including semaphores).

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_NOT_ALLOWED	Arduino Task (Tid = 1) cannot be restarted.
E_TASK_NOT_STARTED	Task not created

EXAMPLE

NAME

Tk_GetActiveTaskNo()

SYNTAX

```
uint8_t Tk_GetActiveTaskNo();
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION

DESCRIPTION

Return the number of active tasks in the system.

RETURNED VALUE

CODE	DESCRIPTION
uint8_t	number of active tasks in the system.

EXAMPLE

NAME

Tk_SetTimeSharing()
Tk_GetTimeSharing()

SYNTAX

```
Bool_t Tk_SetTimeSharing(Bool_t NewValue);
Bool_t Tk_GetTimeSharing();
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
NewValue	IN	TRUE: timesharing enabled FALSE: timesharing disabled

DESCRIPTION

Set the timesharing behavior of uMT and return the previous state. uMT implements a time-sharing functionality: any task is allowed to run only for a fixed maximum time (time slice) before another task is selected to run. The time slice value is initially set by a configuration value (***uMT_TICKS_TIMESHARING***) in uMT (usually 1 second).

RETURNED VALUE

CODE	DESCRIPTION
TRUE	timesharing enabled
FALSE	timesharing disabled

EXAMPLE

NAME

Tk_SetPreemption()
Tk_GetPreemption()

SYNTAX

```
Bool_t Tk_SetPreemption (Bool_t NewValue);
Bool_t Tk_GetPreemption();
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
NewValue	IN	TRUE: preemption enabled FALSE: preemption disabled

DESCRIPTION

Set the pre-emption behavior of uMT and return the previous state. When preemption is disabled, the current task will run until it will call an uMT functionality which can suspend it (e.g., trying to acquire a busy semaphore) or preemption is enabled again. Time sharing is also disabled when preemption is disabled.

The main purpose to disable preemption is to prevent the suspension of the current task still managing interrupts and other interrupt driven system events. Although a good real-time programming style is not relying on disabling preemption, there are cases when this functionality can be needed.

RETURNED VALUE

CODE	DESCRIPTION
TRUE	preemption enabled
FALSE	preemption disabled

EXAMPLE

NAME

Tk_SetBlinkingLED()
Tk_GetBlinkingLED()

SYNTAX

```
Bool_t Tk_SetBlinkingLED(Bool_t NewValue);
Bool_t Tk_GetBlinkingLED();
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
NewValue	IN	TRUE: BUILTIN LED blinking in SysTick enabled FALSE: BUILTIN LED blinking in SysTick disabled

DESCRIPTION

Set the BUILTIN LED behavior of uMT and return the previous state.

The BUILTIN LED (usually input/output #13) is used by uMT in the timer tick routine and in the IDLE task. In the timer tick routing, the BUILTIN LED is turned ON/OFF every second (to give a visual indication that the underlying kernel is working).

IDLE task is setting the LED to ON when it runs (again, to give an easy visual indication that no application task is ready to run).

RETURNED VALUE

CODE	DESCRIPTION
TRUE	BUILTIN LED blinking enabled
FALSE	BUILTIN LED blinking disabled

EXAMPLE

NAME

Tk_SetPriority()
Tk_GetPriority()

SYNTAX

```
Errno_t       Tk_SetPriority(TaskId_t Tid, TaskPrio_t npriority, TaskPrio_t &ppriority);
Errno_t       Tk_GetPriority(TaskId_t Tid, TaskPrio_t &ppriority);
Errno_t       Tk_GetPriority(TaskPrio_t &ppriority);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
Tid	IN	Task ID (if missing, caller task)
Npriority	IN	New priority in the range PRIO_LOWEST (0) to PRIO_HIGHEST(15)
ppriority	OUT	Returned previous priority

DESCRIPTION

uMT tasks can have a priority in the range between 0 (lowest priority task, usually the IDLE task) and 15 (highest priority). NORMAL priority is set to the value of 8 and all new tasks are created with priority equal to NORMAL.

The READY queue is order by task priorities: higher priority tasks go to the head of the queue.

PRIORITY	VALUE

PRIO_LOWEST	0
PRIO_LOW	4
PRIO_NORMAL	8
PRIO_HIGH	12
PRIO_HIGHEST	15
PRIO_MAXPRIO_MASK	0x0F

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success

EXAMPLE

NAME

Tk_SetParam()
Tk_GetParam()

SYNTAX

```
Errno_t       Tk_SetParam(TaskId_t Tid, Param_t _parameter);
Errno_t       Tk_GetParam(Param_t &_parameter);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
Tid	IN	Task ID (if missing, caller task)
_parameter	IN/OUT	Task's parameter

DESCRIPTION

Each task has got an individual parameter launch which can be set between task's creation and task start. This parameter can be read by the launched task at run-time and used accordingly. The parameter type is ***Param_t*** and it can contain any basic data type (including pointers). Currently it is 16 bits on AVR and 32 bits on SAM/SAMD architectures.

The parameter can only be written between ***Kernel.Tk_CreateTask()*** and ***Kernel.Tk_StartTask()*** calls to minimize logic errors.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_ALREADY_STARTED	Task already started, cannot set parameter.

EXAMPLE

NAME

Tk_GetTaskInfo()
Tk_GetTaskInfo()
Tk_PrintInfo()

SYNTAX

```
Errno_t       Tk_GetTaskInfo(TaskId_t Tid, uMTtaskInfo &Info);
Errno_t       Tk_GetTaskInfo(uMTtaskInfo &Info);
Errno_t       Tk_PrintInfo(uMTtaskInfo &Info);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
Tid	IN	Task ID (if missing, caller task)
Info	IN	Structure containing task's information

DESCRIPTION

Task's information are returned in the Info field.

```
class uMTtaskInfo
{
public:
       TaskId_t     Tid;               // Task ID
       TaskPrio_t   Priority;          // Task priority
       Status_t     TaskStatus;        // Task's status
       RunValue_t   Run;               // How many run

       StackSize_t  StackSize;         // Stack's size in bytes
       StackSize_t  FreeStack;         // Free stack size in bytes
       StackSize_t  MaxUsedStack;      // Maximum used stack in bytes

};
```

The Run field contains the number of times this task has been RUNNING.
The MaxUsedStack field contains the maximum stack area used by the task. This heuristic calculation is achieved by setting, at task's creation, the task's stack area to a well-defined value and then scanning the area to discover which part has not been modified yet. The mechanism is not error proof but it gives anyway a good estimate. These two values are also print in the Kn_PrintInternals() call.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success

EXAMPLE

SEMAPHORE MANAGEMENT

Semaphore management provides the following primitives:

NAME	DESCRIPTION
Sm_Claim()	
Sm_Release()	
Sm_SetQueueMode()	
isr_Sm_Claim()	
isr_Sm_Release()	
isr_p_Sm_Release()	

NAME

Sm_Claim()

SYNTAX

```
Errno_t       Sm_Claim(SemId_t Sid, uMToptions_t Options, Timer_t timeout=(Timer_t)0);
Errno_t       Sm_Claim(SemId_t Sid, uMToptions_t Options);
Errno_t       isr_Sm_Claim(SemId_t Sid);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
Sid	IN	Semaphore id, in the range 1-kernelCfg.Semaphores_Num if dynamically configured, 1- uMT_MAX_SEM_NUM if not.
Options	IN	uMT_NOWAIT to return immediately if Semaphore not available. uMT_WAIT otherwise
timeout	IN	Number of milliseconds to wait if Semaphore not available

DESCRIPTION

Try to acquire a semaphore.
When called from ISR, the proper version must be used.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_INVALID_SEMID	Invalid Semaphore ID
E_WOULD_BLOCK	Semaphore not available and uMT_NOWAIT specified as “Options”.
E_TIMEOUT	Timeout expired (Semaphore not taken)

EXAMPLE

NAME

Sm_Release()

SYNTAX

```
Errno_t       Sm_Release(SemId_t Sid);
Errno_t       isr_Sm_Release(SemId_t Sid);
Errno_t       isr_p_Sm_Release(SemId_t Sid);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
Sid	IN	Semaphore id, in the range 1-kernelCfg.Semaphores_Num if dynamically configured, 1- uMT_MAX_SEM_NUM if not.

DESCRIPTION

Release a Semaphore. Task does NOT need to have previously acquired the semaphore.
When called from ISR, the proper version must be used.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_INVALID_SEMID	Invalid Semaphore ID
E_OVERFLOW_SEM	Semaphore value overflow.

EXAMPLE

NAME

Sm_SetQueueMode()

SYNTAX

```
Errno_t       Sm_SetQueueMode(SemId_t Sid, QueueMode_t Mode);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
Sid	IN	Semaphore id, in the range 1-kernelCfg.Semaphores_Num if dynamically configured, 1- uMT_MAX_SEM_NUM if not.
QueueMode	IN	QUEUE_NOPRIO or QUEUE_PRIO

DESCRIPTION

Release a Semaphore. Task does NOT need to have previously acquired the semaphore.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_INVALID_SEMID	Invalid Semaphore ID

E_INVALID_OPTION	Only QUEUE_NOPRIO or QUEUE_PRIO allowed
E_NOT_ALLOWED	Queue mode can only be done if the Semaphore queue is empty

EXAMPLE

EVENT MANAGEMENT

Event management provides the following primitives:

NAME	DESCRIPTION
Ev_Receive()	
Ev_Send()	
isr_Ev_Send()	
isr_p_Ev_Send()	

NAME

Ev_Receive()

SYNTAX

```
Errno_t       Ev_Receive(Event_t   eventin, uMToptions_t flags, Event_t *eventout, Timer_t
       timeout=(Timer_t)0);
Errno_t       Ev_Receive(Event_t   eventin, uMToptions_t flags, Event_t *eventout);
Errno_t       isr_Ev_Receive(Event_t      eventin, Event_t *eventout);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
eventin	IN	Event mask to be received.
flags	IN	Event Mode: uMT_ANY (any event) uMT_ALL (all events are required) Wait Mode (to be ORed with Event Mode) uMT_NOWAIT (do not wait if events are available, return immediately)
eventout	OUT	Events received.
timeout	IN	If not zero, the maximum number of milliseconds to wait

DESCRIPTION

Try to receive the requested Events.
When called from ISR, the proper version must be used. isr_Ev_Receive() is executed with implicit option uMT_NOWAIT and it will never block the caller.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_WOULD_BLOCK	Events not available and uMT_NOWAIT specified as “flags” (eventout is loaded with received events).
E_TIMEOUT	Timeout expired (eventout is loaded with received events).

EXAMPLE

NAME

Ev_Send()

SYNTAX

```
Errno_t       Ev_Send(TaskId_t Tid, Event_t Event);
Errno_t       isr_Ev_Send(TaskId_t Tid, Event_t Event);
Errno_t       isr_p_Ev_Send(TaskId_t Tid, Event_t Event);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
Tid	IN	The task ID which will receive the Event mask.
Event	IN	Event mask to be sent.

DESCRIPTION

Send the Event mask to the specified task Id.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_INVALID_TASKID	Invalid task ID

EXAMPLE

TIMER MANAGEMENT

Timer management provides the following primitives:

NAME	DESCRIPTION
Tm_WakeupAfter()	
Tm_EvAfter()	
Tm_EvEvery()	
Tm_Cancel()	

NAME

Tm_WakeupAfter()

SYNTAX

```
Errno_t Tm_WakeupAfter(Timer_t timeout);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
timeout	IN	Number of milliseconds to wait.

DESCRIPTION

Initialize the uMT subsystem.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success

EXAMPLE

NAME

Tm_EvAfter()

SYNTAX

```
Errno_t Tm_EvAfter(Timer_t timeout, Event_t Event, TimerId_t &TmId);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
timeout	IN	Number of milliseconds (it cannot be zero).

Event	IN	Event mask to sent
TmId	OUT	Timer ID to be used in a subsequent Tm_Cancel().

DESCRIPTION

Send the specified Event mask to the calling task, after a timeout.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_INVALID_TIMEOUT	
E_NOMORE_TIMERS	

EXAMPLE

NAME

Tm_EvEvery()

SYNTAX

```
Errno_t Tm_EvEvery(Timer_t timeout, Event_t Event, TimerId_t &TmId);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
timeout	IN	Number of milliseconds (it cannot be zero).
Event	IN	Event mask to sent
TmId	OUT	Timer ID to be used in a subsequent Tm_Cancel().

DESCRIPTION

Send the specified Event mask to the calling task, every milliseconds.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_INVALID_TIMEOUT	
E_NOMORE_TIMERS	

EXAMPLE

NAME

Tm_Cancel()

SYNTAX

```
Errno_t Tm_Cancel(TimerId_t TmId);
```

PARAMETERS

NAME	IN/OUT	DESCRIPTION
TmId	IN	Timer ID returned by previous Tm_xxx() calls.

DESCRIPTION

Cancel a Timer previously created with a Tm_xxx() call.

RETURNED VALUE

CODE	DESCRIPTION
E_SUCCESS	Success
E_INVALID_TIMERID	
E_NOT_OWNED_TIMER	Timer ID is not owned by the caller task.

EXAMPLE