Getting Started with uMT

This documents provides an introduction to uMT functionality with some basic examples. A reference document for uMT calls 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 v2.6.0 – Doc v2.6.0 – 18 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.

Main functionalities

uMT offers a rich programming environment with over 30 calls:

· ***Task management***: creation and deletion of independent, priority based tasks with a start-up parameter. Moreover, preemption and timesharing can be enabled/disabled at run time.

· ***Semaphore management***: counting semaphores with optional timeout (in the simplest form they can be used as mutual exclusion guards).

· ***Event management***: a configurable number of events per task (16 or 32 events depending on the AVR/SAM architecture) can be used for inter task synchronization, with optional timeout and ALL/ANY optional logic (number of events can be extended to 32/64 by reconfiguration of uMT source code).

· ***Timers management***: task's timers (timeouts) and agent timers (Event generation in future time) are available.

· ***Support Functionalities***: system tick, fatal error, rebooting, etc.

Table of Contents

uMT 1
  Main functionalities 1
Supported Boards 3
Configuration 5
  uMTconfiguration.h 5
  uMTdataTypes.h 5
  Static and Dynamic Configurations 5
Starting uMT 7
Working with Tasks 8
  Yielding Control 9
  Deleting Tasks 9
  Restarting a Task 9
  Tasks Priorities 9
  Tasks Launch Parameter 9
  Time-sharing 10
  Preemption 10
  Task’s Stack Size and Allocation 11
  Task’s Information 11
Working with Events 12
Working with Semaphores 13
Working with Timers 15
  Timer roll-over 15
Return code in the uMT calls 16
Ancillary Functionalities 17
Calling uMT during Interrupt Service Routines (ISR) 18
  Task preemption in ISR management for *isr_XX_YYYY()* calls. 18
  Task preemption in ISR management for *isr_p_XX_YYYY()* calls 18
  ISR management performance 19
Dynamic Memory Management 21
  AVR Platform 21
  SAM/SAMD Platform 21
  Reentrancy 21
System Timer Tick, Rebooting and Built-in LED 22
  AVR architecture 22
  SAM/SAMD architecture 22
  SAMD architecture porting 22
  BUILTIN LED 22
Change Log 23
Author and Contacts 24
Copyright 24

Supported Boards

The main challenge for the Arduino architecture has been the extremely low RAM availability in most of the Arduino boards. For this purpose uMT is configurable both in terms of "subsystems" which can be used (semaphores, events, timers, etc.) and in terms of quantity of objects (tasks, semaphores and timers). As a consequence, uMT is able to run in the 2KB RAM of the ***Arduino Uno*** board although its more natural environment is the ***Arduino Mega2560*** board were the 8KB available RAM allows the design of very sophisticated programs. ***Arduino Due*** is in this context a "luxury" environment thanks to the availability of 96KB of RAM.

The default configuration for ARDUINO Uno is:

Tasks:	5	4 available, including the Arduino main loop()
Semaphores:	8	
Task's Stack size:	200 bytes	To minimize memory usage
Idle task stack size:	128 bytes	To minimize memory usage
Events	16	
Semaphores	8	
Timers	5 task timers 5 agent timers	
TaskRestart functionality	disabled	
Available RAM: (using Test20_Complex1.ino test)	~460 bytes	including Arduino main loop() stack, after Kn_Start() execution.

The default configuration for ARDUINO Mega2560 is:

Tasks:	10	9 available, including the Arduino main loop()
Semaphores:	16	
Task's Stack size:	256 bytes	
Idle task stack size:	128 bytes	
Events	16	
Semaphores	16	
Timers	10 task timers 10 agent timers	
Available RAM: (using Test20_Complex1.ino test)	~4500 bytes	including Arduino main loop() stack, after Kn_Start() execution.

The default configuration for ARDUINO Due is:

Tasks:	20	19 available, including the Arduino main loop()
Semaphores:	32	
Task's Stack size:	1024 bytes	
Idle task stack size:	512 bytes	
Events	32	
Semaphores	32	
Timers	20 task timers 20 agent timers	
Available RAM: (using Test20_Complex1.ino	~70000 bytes	including Arduino main loop() stack, after Kn_Start() execution.

test)		

The default configuration for ARDUINO Zero is:

Tasks:	15	14 available, including the Arduino main loop()
Semaphores:	32	
Task's Stack size:	1024 bytes	
Idle task stack size:	512 bytes	
Events	32	
Semaphores	32	
Timers	15 task timers 15 agent timers	
Available RAM: (using Test20_Complex1.ino test)	Estimated 16000 bytes	Estimated, never tested.

The above configurations can be modified at compile time to increase or decrease requirements (and memory occupation). Moreover, for Arduino Mega2560, Due and Zero boards, a dynamic (run-time) configuration is also possible.

Configuration

uMT can be fully configured at compile-time by modifying the "***uMTconfiguration.h***" and "***uMTdataTypes.h***" files. Moreover, for Arduino Mega2560 and DUE boards, a dynamic configuration (run-time) is also possible.

uMTconfiguration.h

uMT_USE_EVENTS	1 to use Events, 0 otherwise
uMT_USE_SEMAPHORES	1 to use Semaphores, 0 otherwise
uMT_USE_TIMERS	1 to use Timers, 0 otherwise
uMT_USE_RESTARTTASK	1 to use tk_Restart(), 0 otherwise (it saves 4 bytes for each configured task for AVR architecture)
uMT_USE_PRINT_INTERNALS	1 to use tk_Kn_PrintInternals(), 0 otherwise (it saves 30 bytes)
uMT_USE_MALLOC_REENTRANT	1 to use malloc()/realloc()/free() in a thread safe mode by using interrupt locking/unlocking before operating on dynamic memory descriptor list. See also the chapter related to dynamic memory allocation.

Additional, debugging related, configuration is available:

uMT_SAFERUN	default to 1. Only if application has been debugged and one needs additional RAM, this option can be set to 0
uMT_IDLE_TIMEOUT	default to 1. If set, this triggers some special action for IDLE task (e.g., PrintInternals())

Additional configuration can be performed by modifying data types in ***uMTdataTypes.h*** (e.g., Events number and Semaphores maximum counter value).

The uMT environment is implemented as a C++ class (***uMT***) which is already created (instantiated) in the variable ***Kernel***. All the uMT calls are then in the form of ***Kernel.xxx().***

In general, most of the uMT calls return an return code (or E_***SUCCESS*** for successful completion) and it is a good programming practice to test for this exit code (not done in the examples below to improve code readability).

uMTdataTypes.h

The following data types can be changed to fit specific needs.

Param_t	16 bits for AVR, 32 bits for SAM/SAMD.
SemValue_t	16 bits for AVR, 32 bits for SAM/SAMD
uMT_MAX_SEM_VALUE	(0xffff) for 16 bits, (0xffffffff) for 32 bits.

Static and Dynamic Configurations

Arduino Uno can only be configured in a static way using .h files. In Arduino Uno no dynamic memory allocation has been envisioned and .h files must be used.

Arduino Mega2560, Due and Zero boards can have a certain number of parameters configured when calling the ***Kn_Start()*** uMT call. The process requires a call to ***Kn_GetConfiguration()*** to get the current setting, changing what it is needed and then passing the result to the ***Kn_Start()*** call:

```
void setup()
{
	Serial.begin(57600);

	uMTcfg Cfg;

	Kernel.Kn_GetConfiguration(Cfg);

	Cfg.rw.Tasks_Num = MAX_TASKS;
	Cfg.rw.Semaphores_Num = MAX_SEM;
	Cfg.rw.Task1_Stack_Size = 512;
	Cfg.rw.AppTasks_Stack_Size = 512;

	Kernel.Kn_Start(Cfg);

	Kernel.Kn_PrintConfiguration(Cfg);
}
```

The configuration parameters are the following:

PARAMETER	BOARD	DESCRIPTION
Tasks_Num	Mega2560 Due/Zero	Number of the TASKS in the system.
Semaphores_Num	Mega2560 Due/Zero	Number of SEMAPHORES in the system.
AgentTimers_Num	Mega2560 Due/Zero	Number of AGENT TIMERS in the system.
AppTasks_Stack_Size	Mega2560 Due/Zero	Size (in bytes) of the STACK for any new created TASK.
Task1_Stack_Size	Mega2560 Due/Zero	Size (in bytes) of the reserved STACK for the Arduino loop() TASK.
Idle_Stack_Size	Mega2560 Due/Zero	Size (in bytes) of the STACK for the IDLE TASK.
BlinkingLED	All	If TRUE, the SysTick interrupt routine will turn on/off LED 13 every second.
IdleLED	All	If TRUE, when IDLE task is running, LED 13 is turned on.
TimeSharingEnabled	All	If TRUE, Time-Sharing functionality is enabled, FALSE otherwise

When configuration is set dynamically, uMT data area is allocated using ***malloc()*** for SAM architecture, and using ***uMTmalloc()*** for AVR architecture.

Starting uMT

Before any operation, uMT must be started using ***Kernel.Kn_Start()*** call, usually implemented in the Arduino setup() call:

```
void setup()
{
	Serial.begin(57600);

	Serial.println(F("setup(): Initialising..."));
	delay(100); //Allow for serial print to complete.

	Kernel.Kn_Start();
}
```

Although technically speaking uMT could be initialized using a class constructor, the ***Kernel.Kn_Start()*** explicit initialization approach allows a more flexible and complex start up (and initialization) of the whole application.

After ***Kernel.Kn_Start()***, uMT has created 2 tasks: IDLE and ARDUINO.

The IDLE task (task id number 0) is a task which runs when no other task are ready or running. The current version of the IDLE task is setting the BUILTIN LED to ON and, after some time of uninterrupted run, it prints on the Serial the internal status of the uMT kernel. BUILTIN LED functionality is controlled by an option in the ***Kernel.Kn_Start()*** call while printing is controlled by ***uMT_IDLE_TIMEOUT*** setting (and timeout value by ***uMT_IDLE_TIMEOUTVALUE*** setting)

Arduino ***loop()*** routine is uMT task id number 1 and becomes the first RUNNING task in the system.

In general, no operations (e.g., changing task priority) can be performed on the IDLE task.

Working with Tasks

Tasks can be created with the ***Kernel.Tk_CreateTask()*** call and then started (set READY to run) with the ***Kernel.Tk_StartTask()*** call. This two steps strategy allows to create all the required tasks, setting up any required initialization environment and then finally starting them.

The entry point of a new task is specified in the parameter list of ***Kernel.Tk_CreateTask()*** call which returns the Task ID of the newly created tasks. In the second variable, the uMT TASK ID is returned. When ***uMT_VARIABLE_DYNAMIC*** is enabled, an optional third parameters defines the stack size in bytes.

Below an example of task creation:

```
static void Task2()
{
	int counter = 0;
	TaskId_t myTid;

	Kernel.Tk_GetMyTid(myTid);

	Serial.print(F("  Task2(): myTid = "));
	Serial.println(myTid);

	Serial.print(F("  Task2(): Active TASKS = "));
	Serial.println(Kernel.Tk_GetActiveTaskNo());

	Serial.println(F("  Task2(): Deleting myself..."));
	Serial.println(F(""));
	Serial.flush();

	Kernel.Tk_DeleteTask(myTid);

}


void loop()		// TASK TID=1
{
	TaskId_t Tid2;
	TaskId_t Tid3;

	Serial.println(F(" Task1(): Kernel.Tk_CreateTask(Task2)"));

	Kernel.Tk_CreateTask(Task2, Tid2);

// optional  Kernel.Tk_CreateTask(Task2, Tid2, 512);

	Serial.print(F(" Task1(): Task2's Tid = "));
	Serial.println(Tid2);


	Serial.print(F(" Task1(A): Active TASKS = "));
	Serial.println(Kernel.Tk_GetActiveTaskNo());
	Serial.flush();

	Serial.println(F(" Task1(): StartTask(Task2)"));
	Serial.flush();

	Kernel.Tk_StartTask(Tid2);

	Serial.println(F(" Task1(): Yield(A)"));
	Serial.println(F(""));
```

```
        Serial.flush();

        // Run other task
        Kernel.Tk_Yield();

        // Wait for the Tsk1 to die...
        do
        {
                ActiveTasks = Kernel.Tk_GetActiveTaskNo();

                Serial.print(F(" Task1(loop1): Active TASKS = "));
                Serial.println(ActiveTasks);
                Serial.flush();
                delay(500);
        } while (ActiveTasks != 2);

       while (1)
         ;
}
```

Yielding Control

A task can be release the processor and force the execution of the next task in the READY queue (round robin) by calling ***Kernel.Tk_Yield()***. The IDLE task is executed only if no other task is ready to run. If the yielding task is still the highest priority task, no round robin is performed.

Deleting Tasks

A task can be deleted by calling ***Kernel.Tk_DeleteTask()***. A task can also terminate itself by calling this routine. After deletion, the task structure is available again for a new task creation. Arduino loop() task, however, cannot be deleted (because its STACK area cannot be reused in a new task).

Restarting a Task

A task can be restarted by calling ***Kernel.Tk_ReStartTask()***. 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).

Tasks Priorities

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.

Task's priority can be changed with ***Kernel.Tk_SetPriority()*** call to any valid value in the range. Priorities must be used with a great care and a good real-time programming architecture is best based on synchronization and not on priorities. There are cases, however, when priorities are handy and this is supported in uMT.
The READY queue is order by task priorities: higher priority tasks go to the head of the queue.

Tasks Launch Parameter

To increase flexibility, 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.

Below an example:

```
void Producer()
{
	Param_t Param;
	int	HowManyComponents;

	// Get how many
	Kernel.Tk_GetParam(Param);

	int ProdIndex = (int)Param;
	HowManyComponents = ProdIndex + 1;

	< other task activities…>

}


void loop()		// TASK TID=1
{
	TaskId_t TidProducer[MAX_PRODUCERS];


	for (int idx = 0; idx < MAX_PRODUCERS; idx++)
	{
		Kernel.Tk_CreateTask(Producer, TidProducer[idx]);

		// Set parameters for Producers
		Kernel.Tk_SetParam(TidProducer[idx], idx);
	}

	// Start tasks
	for (int idx = 0; idx < MAX_PRODUCERS; idx++)
		Kernel.Tk_StartTask(TidProducer[idx]);

	< other task activities…>

}
```

Time-sharing

uMT also 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). Time sharing can be disabled/enabled run-time calling ***Kernel.Tk_SetTimeSharing()*** or by an option in the ***Kernel.Kn_Start()*** initial call.

Preemption

The preemptive behavior of uMT can be enabled and disabled at run-time by calling ***Kernel.Tk_SetPreemption()***. 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.

Task's Stack Size and Allocation

When ***uMT_VARIABLE_DYNAMIC*** is set for a specific board, a task's stack size (in bytes) can be specified in the ***Tk_CreateTask()*** call. This is giving a much greater flexibility and control to how much memory is used.

When ***uMT_VARIABLE_DYNAMIC*** is set, uMT is allocating stack area by using ***malloc()*** for SAM architecture, and using ***uMTmalloc()*** for AVR architecture.

Board	Default Allocation Type
Arduino Uno	uMT_FIXED_STATIC (do NOT change!)
Arduino Mega2560	uMT_VARIABLE_DYNAMIC
Arduino Due/Zero	uMT_VARIABLE_DYNAMIC

Default values are defined in "uMTconfiguration.h" file.

Task's Information

It is possible to get task's information using the ***Tk_GetTaskInfo()*** (for the caller or any task in the system) and print it on Serial by calling ***Tk_PrintInfo()***.

```
////////////////////////////////////////////////////
// Returned in Tk_GetTaskInfo()
////////////////////////////////////////////////////

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 inbytes
	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.

Working with Events

Events are a set of binary flags related to each task which can be set by the owner task or by other tasks. They provide a very basic but also very powerful synchronization mechanism between tasks. Event setting can also be executed during Interrupt Service Routines (ISR).

Each task is equipped with 16 or 32 Events (depending on uMT configuration, typically 16 for AVR and 32 for SAM architectures, which can be independently reconfigured in uMT).

Kernel.Ev_Send() primitive is available to set one or more Events and ***Kernel.Ev_Receive()*** primitive is available to wait for Events to happen. In the wait primitive it is possible to specify ***uMT_ANY*** to be awoken if at least one has been set. ***uMT_NOWAIT*** can be optionally specified to return control immediately if the event condition is not met (and current event set is returned as well).

Optionally (if ***TIMERS*** are enabled) a timeout can be specified to avoid to wait longer than a predefined time.

Below an example:

```
#define EVENT_A     0x0001
#define EVENT_B     0x0002

void Task2()
{
	int counter = 0;
	TaskId_t myTid;

	Kernel.Tk_GetMyTid(myTid);

	Serial.print(F("  Task2(): myTid = "));
	Serial.println(myTid);
	Serial.flush();

	while (1)
	{
		Serial.println(F("  Task2(): iEv_Send(1, EVENT_A)"));
		Serial.flush();

		Kernel.Ev_Send(1, EVENT_A);

		Serial.println(F("  Task2(): iEv_Receive(EVENT_B, uMT_ANY)"));
		Serial.flush();

		Kernel.Ev_Receive(EVENT_B, uMT_ANY, &eventout);

		Serial.println(F("  Task2(): delay(3000)"));
		Serial.flush();
		delay(3000);
	}
}
```

See the examples in the uMT library for further details.

Working with Semaphores

uMT provides counting Semaphores for highly sophisticated inter-task communication.

A Semaphore is a synchronization object that controls access by multiple processes to a common resource in a parallel programming environment.

The simplest form of a Semaphore provides only a binary value: free (1) or busy (0). uMT implementation, however, provides counting semaphores to allow a more flexible inter-task communication (using the value of 0 for a “busy” semaphore which is the initial semaphore value as well).

The maximum value of a counting Semaphore is configuration dependent (16 bits for AVR and 32 bits for SAM, which can be reconfigured at run time (at the expenses of additional RAM memory space).

Semaphore queues (that is the list of tasks blocked in the attempt to acquire a busy semaphore) can be ordered by task priority (default uMT behavior). If this is undesirable, each individual Semaphore queue can be configured differently by calling ***Sm_SetQueueMode()***, usually before Semaphore first utilization.

Below an example of use of Semaphores:

```
#define      SEM_ID_01           1             // Semaphore id
#define      SEM_ID_02           2             // Semaphore id

void Task2()
{
       int counter = 0;
       TaskId_t myTid;

       Kernel.Tk_GetMyTid(myTid);

       Serial.print(F("  Task2(): myTid = "));
       Serial.println(myTid);
       Serial.flush();

       while (1)
       {
              Serial.println(F("  Task2(): iSm_Release(SEM_ID_01)"));
              Serial.flush();

              Kernel.Sm_Release(SEM_ID_01);

              Serial.println(F("  Task2(): iSm_Claim(SEM_ID_02)"));
              Serial.flush();

              Kernel.Sm_Claim(SEM_ID_02, uMT_WAIT);

              Serial.println(F("  Task2(): Tm_WakeupAfter(4000)"));
              Serial.flush();
              Kernel.Tm_WakeupAfter(4000);

              Serial.println(F("  Task2(): kicking again..."));
              Serial.flush();

       }
}
```

```
void loop()         // TASK TID=1
{
        int counter = 0;
        Errno_t error;
        TaskId_t Tid;

        Serial.println(F(" Task1(): Kernel.Tk_CreateTask(Task1)"));

        Kernel.Tk_CreateTask(Task2, Tid);

        Serial.print(F(" Task1(): Task2's Tid = "));
        Serial.println(Tid);

        Serial.println(F(" Task1(): StartTask(Task1)"));

        Kernel.Tk_StartTask(Tid);

        TaskId_t myTid;
        Kernel.Tk_GetMyTid(myTid);

        Serial.print(F(" Task1(): myTid = "));
        Serial.println(myTid);

        while (1)
        {
                Serial.print(F(" Task1(): iSm_Claim(SEM_ID_01), timeout=1000"));
                Serial.flush();

                Timer_t timeout = 1000;   // Timeout 1 second (other task, 3 seconds)

                while ( (error = Kernel.Sm_Claim(SEM_ID_01, uMT_WAIT, timeout)) != E_SUCCESS)
                {
                        if (error == E_TIMEOUT)
                        {
                                // Timeout
                                Serial.println(F("Task1(): Sm_Claim(): timeout!"));
                                timeout = (Timer_t)5000;  // Large timeout...
                        }
                        else
                        {
                                Serial.print(F("Task1(): Sm_Claim() Failure! - returned "));
                                Serial.println((unsigned)error);
                                delay(5000);
                        }
                }

                Serial.print(F(" Task1(): iSm_Release(SEM_ID_02)"));
                Kernel.Sm_Release(SEM_ID_02);
        }
}
```

Working with Timers

uMT provides timers to wait for a specified number of milliseconds (without using system resources) or to send Events is a future time possibly in an automatic repeated mode.

***Kernel.Tm_WakeupAfter(***) can be used to wait for a defined number of milliseconds.

Kernel.Tm_EvAfter() can be used to send an Event after a defined number of milliseconds.

Kernel.Tm_EvEvery() can be used to send an Event every a defined number of milliseconds.

Timers can also be cancelled with ***Kernel.Tm_Cancel()*** call.

Multiple timers can be activated for the same task until Timers resource availability has terminated.

uMT is configured with one timer (called TASK TIMER) for each task to support ***Kernel.Tm_WakeupAfter()*** [this approach guarantees that this call will never fail for resource unavailability, at expenses of additional RAM] and with a number of AGENT TIMER defined in the uMT configuration file (default equal to the total number of tasks). For example, in a configuration with 10 tasks a total of 20 Timers (10 TASK TIMERS + 10 AGENT TIMERS) are available.

A detailed example of Timers use is available in the example folder of the uMT library.

Timer roll-over

The Arduino standard SysTick management is based on a 32 bits counter (as returned by millis() Arduino call). This counter is counting milliseconds and it is rolling over after roughly 50 days (4,294,967,295 / 1000 /3600 / 24 = 49,7 days).
uMT is implementing its internal tick counter with 48 bits (in ***uMTextendedTime.h***) extending the total roll-over to almost 8900 years. uMT calls, however, limit the timeout value to 32 bits limiting the range for any future event to 50 days. Because uMT is managing the timer roll-over, the timeout is properly managed also when any 50 days roll-over is achieved.

As a conclusion:
1. Any future event cannot be farther than 49,7 days (32 bit)
2. This 49,7 days window can be applied in the full 8900 years period window.
3. A repeated event (e.g. ***Kernel.Tm_EvEvery()***) can be working for the full 8900 years period if the delays is below 50 days.

Return code in the uMT calls

In general all the uMT calls return an return code (or ***E_SUCCESS*** for successful completion). It is a good programming practice to test for this exit code (not done in the examples listed in this document, to improve code readability).

Current return codes (see “***uMTerrno.h***”):

```
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()]
};
```

Ancillary Functionalities

Additional helper functionalities are available in uMT. Here a list with a short description (look at ***uMT*** class definition for a full list).

CALL	DESCRIPTION
isr_Kn_FatalError()	Print an optional string to Serial and the calls ***isrKn_Reboot()***
isr_Kn_Reboot()	Reboot the system
isr_Kn_GetVersion()	Return uMT version number
Kn_PrintInternals()	Print on Serial some internal uMT data structure values (task list, semaphore list, timer list, etc.)
Isr_Kn_GetKernelTick()	Return the uMT tick counter (usually equal to millis() call)
Tk_SetBlinkingLED()	Control if the BUILTIN LED if used by IDLE task and timer tick routines (uMT_AVR_wiring.c)
isr_Kn_IntLock()	Disable interrupts and returns previous interrupt mask to be used in subsequent ***isrKn_IntUnlock().***
isr_Kn_IntUnlock()	Restore previous interrupt mask
Tk_GetTaskInfo()	Obtain task information
Tk_PrintInfo()	Print on Serial the task information previously obtained with Tk_GetTaskInfo()

Calling uMT during Interrupt Service Routines (ISR)

Some uMT primitives can also be called from ISR. Their name is in the form of ***isr_XX_YYYY()*** or ***isr_p_XX_YYYY()***.

Here the current list:

CALL
isr_Kn_FatalError()
isr_Kn_Reboot()
isr_Kn_GetVersion()
isr_Kn_GetKernelTick()
isr_Kn_IntLock()
isr_Kn_IntUnlock(CpuStatusReg_t Flags)
isr_Sm_Release(SemId_t Sid)
isr_p_Sm_Release(SemId_t Sid)
isr_Sm_Claim(SemId_t Sid)
isr_Ev_Send(TaskId_t Tid, Event_t Event)
isr_p_Ev_Send(TaskId_t Tid, Event_t Event)
isr_Ev_Receive(Event_t eventin, Event_t *eventout);

Task preemption in ISR management for *isr_XX_YYYY()* calls.

To improve compatibility and simplify ISR development, calls to uMT isr_XX_YYYY() primitives will NOT trigger a reschedule and will NOT preempt current caller routines (so control is always returned to caller routine after uMT functionality completion).

If a call to uMT ISR ***isr_XX_YYYY()*** primitive is generating a logical preemption (e.g., higher priority task becoming ready after Event delivery), this is either managed:

1. In Idle loop by triggering a Yield()
2. In the TimerTicks by triggering a reschedule().

See the dedicated section with measurements.

Task preemption in ISR management for *isr_p_XX_YYYY()* calls

These calls will preempt the caller if the awaked task has an higher priority of the current running task.

There is a implementation difference between AVR and SAM architecture:

· On AVR architecture, the caller must complete all the interrupt required steps before calling ***isr_p_XX_YYYY()*** because the latter might not return control.

· On SAM/SAMD architecture, because task switching is only performed using ***pendSVHook()*** exception processing, the caller routine will complete its interrupt processing if its interrupt priority is higher than ***pendSVHook*** priority (uMT is initializing ***pendSVHook*** priority at lowest possible level). As a consequence, it is suggested to always use ***isr_p_XX_YYYY()*** because of their faster performance.

ISR management performance

The “Test30_InterruptLatency.cpp” test has been designed to measure interrupt latency when using direct interrupt handler (no uMT) and using uMT Events (***isr_*** and ***isr_p_*** calls).

Times are in microseconds.

Board type	Measurement Type	No uMT	isr_Ev_Send()	isr_p_Ev_Send()
Uno	micros()	8/12	64	20/24
Mega2560	micros()	12/16	68	24/28
Due	micros()	4	22	20
Due	SysTick->VAL	2	21	18

On AVR boards, the interrupt is generated by setting high a Port (PIN) every second. Note that on AVR boards, micros() has a resolution of 4 microseconds.

On Due board, a timer is used to generate periodic interrupts.

Please note that this is a best case scenario when not very many interrupts are executing at the same time. For SAM/SAMD boards, the ***pendSVHook*** interrupt is set to the lowest level so a task switching will occur only when all pending interrupts have been serviced.
As a final consideration, direct interrupt handling is significantly faster than uMT Event management. It is then a trade-off between exceptional performance (direct) and high-performance with rich functionalities (uMT) and it is up to the user to decide what it will fit best.

For both architecture, the overall software schema for latency measurement is the following:

```
static void interruptHandler()
{
	time0 = MICROS();
	digitalWrite(interruptPin, LOW);

	if (ISR_mode == 0)
		return;

	if (ISR_mode == 1)
		Kernel.isr_Ev_Send(ARDUINO_TID, EVENT_A);
	else
		Kernel.isr_p_Ev_Send(ARDUINO_TID, EVENT_A);
}




void LoopWithoutKernel(int idx)  // ISR_Mode 0
{
	unsigned long elapsed;
	time0 = 0L;

	while (time0 == 0)
		;

	// Read the timer
	elapsed = MICROS() - time0;


	Serial.print(F("No uMT: counter = "));
	Serial.print(idx);
	Serial.print(F(" - Elapsed micros = "));
	Serial.println(elapsed);
	Serial.flush();
}




static void Receiver()		// ISR_Mode 1 and 2
{
	unsigned counter = 0;

	while (1)
	{
		SystemTick_t elapsed;
		Event_t      eventout;

		Kernel.Ev_Receive(EVENT_A, uMT_ANY, &eventout);

		// Read the timer
		elapsed = MICROS() - time0;

		Serial.print(F("Receiver(): counter = "));
		Serial.print(counter++);
		Serial.print(F(" - Elapsed micros = "));
		Serial.println(elapsed);
		Serial.flush();

	}
}
```

Dynamic Memory Management

To enhance flexibility and to provide compatibility with ***new/delete*** C++ mechanisms (and malloc(), realloc(), free() for C), uMT is providing extensive support for dynamic memory allocation.

On the AVR platform, the standard dynamic memory allocation is using the Arduino ***loop()*** stack pointer as an upper limit for Heap management. This is very inconvenient because the stacks of all the newly created tasks will be locate below the heap area, triggering a failure in the malloc() call. uMT implementation (or configuration) of dynamic memory allocation is avoiding this problem and also providing thread safe functionality.

AVR Platform

Ideally, uMT would need to replace the malloc()/realloc()/free() calls with its own version which are using a thread safe approach. Unfortunately in current uMT version the Author has been unable (!) to replace them from the AVR library so a different schema is used, as explained below.

AVR malloc() can be configured by setting the 2 variables: "__malloc_heap_start" and __malloc_heap_end" to the START and to the END of the memory area used for dynamic memory management. This setting must be performed before the first call to malloc(). However, because we only need to change the end of the range (__malloc_heap_end) this can be done afterwards. As a consequence, in ***Kn_Start()*** the "__malloc_heap_end" is set to the end of the RAM (RAMEND) minus the size of the Arduino loop() stack size as defined in the uMT configuration (static or dynamic).

On the AVR platform, the standard malloc()/free() routines (dynamic memory allocation) are not using at the moment a thread safe approach. To achieve the same result, uMT is providing a new set of calls which will be fully reentrant still calling the original malloc()/realloc()/free(). The c++ ***new*** operator has been redefined to take advantage of the new routines.

uMT CALL	AVR equivalent
uMTmalloc	malloc()
uMTrealloc()	realloc()
uMTfree()	free()

Reentrance is controlled by the `uMT_USE_MALLOC_REENTRANT` configuration flag.

It is then advised to redefine malloc/realloc/free with the uMT version (e.g., with a #define in a proper place of selected .h files).

SAM/SAMD Platform

On the SAM/SAMD platform, the standard malloc()/free() routines have been replaced with the equivalent coming from the AVR library. These routines are fully reentrant (provided that `uMT_USE_MALLOC_REENTRANT` configuration flag is set to 1).

Reentrancy

Thread safe is achieved by encapsulating malloc/realloc/free functionality with ***Kernel.isr_Kn_IntLock()*** and ***Kernel.isr_Kn_IntUnlock()*** calls. This is the safest approach although it is sacrificing real-time response in the system.

System Timer Tick, Rebooting and Built-in LED

AVR architecture

For AVR architecture, uMT is relying on the standard TIMER0 system tick to generate its base system tick. For this reason the standard "***wiring.c***" file has been modified (in ***"uMT_AVR_wiring.c"***) and new code has been developed to implement preemption, timesharing and Timers management (in ***"uMT_AVR_SysTick.cpp"***).

For AVR architecture, the Watchdog timer (WDT) is used to generate a system reboot (in ***isr_Kn_Reboot()***).

SAM/SAMD architecture

For SAM/SAMD architecture, uMT is relying on the standard SysTick management by providing a ***sysTickHook()*** implementation and it is using ***pendSVHook()*** exception processing to perform task switching.

For SAM/SAMD architecture, the ***NVIC_SystemReset(***) call is used to generate a system reboot (in ***isr_Kn_Reboot()***).

SAMD architecture porting

An adaptation for SAMD architecture (Arduino Zero) has been performed however the board has not been tested (not being available).

The modifications for SAMD are:

1. On SAMD, PUSH/POP operations are limited to R0-R7 registers. So a different PUSH/POP instruction set have been used (this has been tested with SAM board so it should work).
2. On SAMD, memory size is limited to 32K (this has been modified but not tested).
3. On SAMD, NVIC interrupt priority is limited to 32 (this has been modified but not tested).
4. On SAMD, millis() is used to get current SysTick (***GetTickCount()*** on SAM). Not tested.
5. On SAMD, the symbol "__end__" has been used for start of Heap for malloc(). SAM is using "_end". Not tested.
6. Arduino IDE correctly compile the uMT software and all the tests when set as SAMD (Arduino Zero) board.

Overall, the Author does expect the Zero board to work but, again, it has not been tested.

BUILTIN LED

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).

Both functions can be disabled in the configuration file at compile or run time.

Change Log

Version 2.6.0 – 18 June 2017

1. Implemented uMTobject_id class to protect from misuse/use of deleted object. uMTobject class now used for TimerId and TaskId.
2. Implemented task running timer counter to track elapsed time spent by tasks in S_RUNNING state.
3. TimeTicks resolution upgraded to 48 bits (from 40 bits). Now timer can span almost 9000 years.

Version 2.5.0 – 07 June 2017

4. Added support for Arduino Zero (SAMD - Atmel ARM Cortex-M0 CPU) board.
5. Measured interrupt latency.
6. Added Task info and task's run counter

Version 2.0.0 – June 2017

7. Added full support for Arduino DUE (SAM - Atmel SAM3X8E ARM Cortex-M3 CPU) board.
8. Added support for isrp_XX_YYYY() calls (ISR level uMT calls with task preemption).
9. Fixed Sm_Release() parameter list (it was incorrectly never preempting the calling task).
10. Revised lock/unlock critical region strategy (hopefully now more robust…)
11. Implemented Static and Dynamic Configurations
12. Implemented dynamic memory allocation for uMT objects and tasks' stacks.
13. Implemented re-entrant malloc()/realloc()/free()

Version 1.5.0 – May 2017

1. Initial public release

Author and Contacts

Author: Antonio Pastore - Italy
Contact: go0126@alice.it

Copyright