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 SW 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.

uMT version 1.5.0 - 22 May 2017

uMT

Micro Multi Tasker (uMT) is a preemptive, timesharing, real-time 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 architecture (Arduino DUE) without Timers functionality (a full port is planned for a future version).

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:

- ***Task management***: creation of independent, priority based, tasks with starting 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) can be used for inter task synchronization, with optional timeout and ALL/ANY optional logic (number of events can be extended 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.

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 96KB RAM available.

The standard 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:	*96 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)*	*551 bytes*	*including Arduino main loop() stack*

The standard 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:	*96 bytes*	
Events	*16*	
Semaphores	*16*	
Timers	*10 task timers* *10 agent timers*	
Available RAM: *(using Test20_Complex1.ino test)*	*4785 bytes*	*including Arduino main loop() stack*

The standard 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:	*1024 bytes*	
Events	*32*	
Semaphores	*32*	
Timers	*20 task timers* *20 agent timers*	
Available RAM: *(using Test00_PrintConfiguration test)*	*71440 bytes*	*including Arduino main loop() stack*

The above configurations can be modified at compile time to increase or decrease requirements (and memory occupation).

Configuration

uMT can be fully configured by modifying the "uMTconfiguration.h" and “uMTdataTypes.h” files.

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)

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

Starting uMT

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

```
void setup()
{
	// put your setup code here, to run once:

	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.

ARDUINO task (task id number 1) is the ***loop()*** Arduino task which becomes the first RUNNING task in the system.

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.

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);
```

```
	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 by calling ***Kernel.Tk_Yield()***. The IDLE task is executed only if no other task is ready to run.

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.

Restarting a Task

A task can be restarted by calling ***Kernel.Tk_ReStartTask()***. For memory space reasons (it would have required the allocation of a private uMT stack), a task cannot restart itself (however, the restart functionality can be programmed in the application task itself).
A task restart will reset the initial stack pointer to the entry routine of the task but it will not change current task priority. It remains in 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 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.

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();

      Timer_t OldSMillis = millis();

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

            Kernel.Ev_Send(1, EVENT_A);

            Timer_t NowMillis = millis();
            Timer_t DeltaMillis = NowMillis - OldSMillis;
            OldSMillis = NowMillis;

            Serial.print(F("  Task2(): => "));
            Serial.print(++counter);
            Serial.print(F("  Delta in millis() => "));
            Serial.println(DeltaMillis);

            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 more 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.

Usually 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 order 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();

      Timer_t OldSMillis = millis();

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

            Timer_t NowMillis = millis();
            Timer_t DeltaMillis = NowMillis - OldSMillis;
            OldSMillis = NowMillis;

            Serial.print(F("  Task2(): => "));
            Serial.print(++counter);

            Serial.print(F("  Delta in millis() => "));
            Serial.println(DeltaMillis);

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

            Kernel.Sm_Claim(SEM_ID_02, uMT_WAIT);


#ifdef USE_DELAY
            Serial.println(F("  Task2(): delay(3000)"));
            Serial.flush();
            delay(4000);
#else
            Serial.println(F("  Task2(): Tm_WakeupAfter(4000)"));
            Serial.flush();
            Kernel.Tm_WakeupAfter(4000);
#endif
            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)"));
      Serial.flush();

      Kernel.Tk_StartTask(Tid);

      TaskId_t myTid;

      Kernel.Tk_GetMyTid(myTid);

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

      while (1)
      {
            Serial.print(F(" Task1("));
            Serial.print(millis());
            Serial.println(F("): 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.print(F(" Task1("));
                        Serial.print(millis());
                        Serial.println(F("): Sm_Claim(): timeout!"));
                        Serial.flush();

                        timeout = (Timer_t)5000;      // Large timeout...

                        Serial.print(F(" Task1("));
                        Serial.print(millis());
                        Serial.println(F("): iSm_Claim(SEM_ID_01), timeout=5000"));
                        Serial.flush();

                  }
                  else
                  {
                        Serial.print(F(" Task1("));
                        Serial.print(millis());
                        Serial.print(F("): Sm_Claim() Failure! - returned "));
                        Serial.println((unsigned)error);
                        Serial.flush();

                        delay(5000);

                  }
            }


            Serial.print(F(" Task1("));
            Serial.print(millis());
            Serial.print(F("): => "));
            Serial.print(++counter);

            Serial.print(F(" Task1("));
            Serial.print(millis());
            Serial.println(F("): iSm_Release(SEM_ID_02)"));
            Serial.flush();

            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.

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:

```
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_TOTAL_TASK_NUM,  // Not enough configured TASKs(Kn_Start)
/* 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};
};
```

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

`isrKn_FatalError()`	Print an optional string to Serial and the calls Kn_iReboot()
`isrKn_Reboot()`	Reboot the system
`isrKn_GetVersion()`	Return uMT version number
`Kn_PrintInternals()`	Print on Serial some internal uMT data structure values (task list, semaphore list, timer list, etc.)
`isrKn_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)
`isrKn_IntLock()`	Disable interrupts and returns previous interrupt mask to be used in subsequent isrKn_IntUnlock().
`isrKn_IntUnlock()`	Restore previous interrupt mask

Calling uMT during Interrupt Service Routines (ISR)

Few of the uMT primitives can be called from ISR. Their name are in the form of isrXX_YYYY().

Here the current list:

CALL
`isrKn_FatalError()`
`isrKn_Reboot()`
`isrKn_GetVersion()`
`isrKn_GetKernelTick()`
`isrKn_IntLock()`
`isrKn_IntUnlock(CpuStatusReg_t Flags)`
`isrSm_Release(SemId_t Sid)`
`isrSm_Claim(SemId_t Sid)`
`isrEv_Send(TaskId_t Tid, Event_t Event)`

Task preemption in ISR management

To improve compatibility and simplify ISR development, calls to uMT isrXX_YYY 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 isrXX_YYY 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().

As a consequence, the maximum delay is less than TimerTicks interval (currently 1 millisecond).

Dynamic Memory Management

On the AVR platform, the standard malloc()/free() functions (dynamic memory allocation) are using, as dynamic memory allocation, any memory available between:

A. the end of the compile time defined data area (allocated and unallocated) and
B. the current stack pointer value minus a “safe” size.

As a consequence, malloc() uses the current task’s stack pointer to determine free available memory and assumes that the stack pointer is referring to the Arduino loop() routine.

If a uMT stack is allocated statically, a call to malloc() from the task using this stack will fail (because the stack is in below the minimum RAM address used by malloc()).

To avoid malloc() to fail, uMT is allocating all stacks (except the IDLE task stack, which never calls malloc()) in the high memory region. The top most slot is reserved for the Arduino loop() task whose stack size is assumed to be the same as the other uMT tasks (but this can be reconfigured at compile time).

An exception is ARDUINO UNO platform where all stacks are allocated statically (it makes no sense to use malloc() in a 2KB RAM environment).

The above behavior is controlled by the ***uMT_STATIC_STACKS*** define (1=static, 2=reallocated in the top memory). In case it not wanted to allocate stack memory dynamically, the above ***configuration*** can be set to 0.

As a general note, it is recommended to avoid dynamic memory allocation and use only compile time (static or uninitialized) data area, unless a large memory area is available (as in the DUE). This is a best practice because of the risk of malloc() to fail in systems with such a limited memory amount (note that some MIL type specifications prohibit dynamic memory allocation because of this).

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 ***isrKn_Reboot()***).

SAM architecture

For SAM architecture, uMT is relying on the standard SysTick management by providing a ***sysTickHook()*** Implementation. In this version of uMT a full TimerTicks implementation is not available yet (porting not finished) and Timesharing or Timers are not available.

For SAM architecture, a specialized instruction is used to generate a system reboot (in ***isrKn_Reboot()***).

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).

Change Log

Version 1.5.0
Initial public release

Author and Contacts

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

Copyright

uMT – a multi tasker for the Arduino platform