Task Creation

Modules

- xTaskCreate
- xTaskCreateStatic
- vTaskDelete

Detailed Description

TaskHandle_t

task. h

Type by which tasks are referenced. For example, a call to xTaskCreate returns (via a pointer parameter) an TaskHandle_t variable that can then be used as a parameter to vTaskDelete to delete the task

xTaskCreate

[Task Creation]

task. h

```
 BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
                            const char * const pcName,
                            unsigned short usStackDepth,
                            void *pvParameters,
                            UBaseType_t uxPriority,
                            TaskHandle_t *pxCreatedTask
                          );
```

Create a new task and add it to the list of tasks that are ready to run. ConfigSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or left undefined (in which case it will default to 1), for this RTOS API function to be available.

Each task requires RAM that is used to hold the task state, and used by the task as its stack. If a task is created using xTaskCreate () then the required RAM is automatically allocated from the FreeRTOS heap. If a task is created using xTaskCreateStatic() then the RAM is provided by the application writer, so can be statically allocated at compile time. See the Static Vs Dynamic allocation page for more information.

If you are using FreeRTOS-MPU then it is recommended to use xTaskCreateRestricted() in place of xTaskCreate().

Parameters:

pvTaskCode	Pointer to the task entry function (just the name of the function that implements the task, see the example below). Tasks are normally implemented as an infinite loop, and must never attempt to return or exit from their implementing function. Tasks can however delete themselves.
pcName	A descriptive name for the task. This is mainly used to facilitate debugging, but can also be used to obtain a task handle. The maximum length of a task's name is set using the configMAX_TASK_NAME_LEN parameter in FreeRTOSConfig.h.
usStackDepth	The number of words (not bytes!) to allocate for use as the task's stack. For example, if the stack is 16-bits wide and usStackDepth is 100, then 200 bytes will be allocated for use as the task's stack. As another example, if the stack is 32-bits wide and usStackDepth is 400 then 1600 bytes will be allocated for use as the task's stack. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t.
pvParameters	A value that will passed into the created task as the task's parameter. If pvParameters is set to the address of a variable then the variable must still exist when the created task executes - so it is not valid to pass the address of a stack variable.
uxPriority	The priority at which the created task will execute. Systems that include MPU support can optionally create a task in a privileged (system) mode by setting bit portPRIVILEGE_BIT in uxPrriority. For example, to create a privileged task at priority 2 set uxPriority to (2 | portPRIVILEGE_BIT).
pxCreatedTask	Used to pass a handle to the created task out of the xTaskCreate() function. pxCreatedTask is optional and can be set to NULL.

Returns:

If the task was created successfully then pdPASS is returned. Otherwise errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY is returned.

Example usage:

```
/* Task to be created. */
void vTaskCode( void * pvParameters )
{
    /* The parameter value is expected to be 1 as 1 is passed in the
    pvParameters value in the call to xTaskCreate() below.
    configASSERT( ( ( uint32_t ) pvParameters ) == 1 );

    for( ;; )
    {
        /* Task code goes here. */
    }
}

/* Function that creates a task. */
void vOtherFunction( void )
{
BaseType_t xReturned;
TaskHandle_t xHandle = NULL;

    /* Create the task, storing the handle. */
    xReturned = xTaskCreate(
                    vTaskCode,       /* Function that implements the task. */
                    "NAME",          /* Text name for the task. */
                    STACK_SIZE,      /* Stack size in words, not bytes. */
                    ( void * ) 1,    /* Parameter passed into the task. */
                    tskIDLE_PRIORITY,/* Priority at which the task is
created. */
                    &xHandle );      /* Used to pass out the created task's
handle. */

    if( xReturned == pdPASS )
    {
        /* The task was created.  Use the task's handle to delete the task.
*/
        vTaskDelete( xHandle );
    }
```

xTaskCreateStatic

[Task Creation]

```
task. h
 TaskHandle_t xTaskCreateStatic( TaskFunction_t pxTaskCode,
                                 const char * const pcName,
                                 const uint32_t ulStackDepth,
                                 void * const pvParameters,
                                 UBaseType_t uxPriority,
                                 StackType_t * const puxStackBuffer,
                                 StaticTask_t * const pxTaskBuffer );
```

Create a new task and add it to the list of tasks that are ready to run. configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h for this RTOS API function to be available.

Each task requires RAM that is used to hold the task state, and used by the task as its stack. If a task is created using xTaskCreate() then the required RAM is automatically allocated from the FreeRTOS heap. If a task is created using xTaskCreateStatic() then the RAM is provided by the application writer, which results in a greater number of parameters, but allows the RAM to be statically allocated at compile time. See the Static Vs Dynamic allocation page for more information.

If you are using FreeRTOS-MPU then it is recommended to use xTaskCreateRestricted() in place of xTaskCreateStatic().

Parameters:

pvTaskCode	Pointer to the task entry function (just the name of the function that implements the task, see the example below). Tasks are normally implemented as an infinite loop, and must never attempt to return or exit from their implementing function. Tasks can however delete themselves.
pcName	A descriptive name for the task. This is mainly used to facilitate debugging, but can also be used to obtain a task handle. The maximum length of a task's name is set using the configMAX_TASK_NAME_LEN parameter in FreeRTOSConfig.h.
ulStackDepth	The puxStackBuffer parameter is used to pass an array of StackType_t variables into xTaskCreateStatic(). ulStackDepth must be set to the number of indexes in the array. See the FAQ How big should the stack be?
pvParameters	A value that will passed into the created task as the task's parameter. If pvParameters is set to the address of a variable then the variable must still exist when the created task executes - so it is not valid to pass the address of a stack variable.
uxPriority	The priority at which the created task will execute. Systems that include MPU support can optionally create a task in a privileged (system) mode by setting bit portPRIVILEGE_BIT in uxPrriority. For example, to create a privileged task at priority 2 set uxPriority to (2 | portPRIVILEGE_BIT).
puxStackBuffer	Must point to a StackType_t array that has at least ulStackDepth

indexes (see the ulStackDepth parameter above) - the array will be used as the task's stack, so must be persistent (not declared on the stack of a function).

pxTaskBuffer	Must point to a variable of type StaticTask_t. The variable will be used to hold the new task's data structures (TCB), so it must be persistent (not declared on the stack of a function).

Returns:

If neither puxStackBuffer or pxTaskBuffer are NULL then the task will be created, and the task's handle is returned. If either puxStackBuffer or pxTaskBuffer is NULL then the task will not be created and NULL will be returned.

Example usage:

```
    /* Dimensions the buffer that the task being created will use as its 
stack.
    NOTE:  This is the number of words the stack will hold, not the number of
    bytes.  For example, if each stack item is 32-bits, and this is set to 
100,
    then 400 bytes (100 * 32-bits) will be allocated. */
    #define STACK_SIZE 200

    /* Structure that will hold the TCB of the task being created. */
    StaticTask_t xTaskBuffer;

    /* Buffer that the task being created will use as its stack.  Note this 
is
    an array of StackType_t variables.  The size of StackType_t is dependent 
on
    the RTOS port. */
    StackType_t xStack[ STACK_SIZE ];

    /* Function that implements the task being created. */
    void vTaskCode( void * pvParameters )
    {
        /* The parameter value is expected to be 1 as 1 is passed in the
        pvParameters value in the call to xTaskCreateStatic(). */
        configASSERT( ( uint32_t ) pvParameters == 1UL );

        for( ;; )
        {
            /* Task code goes here. */
        }
    }

    /* Function that creates a task. */
    void vOtherFunction( void )
    {
        TaskHandle_t xHandle = NULL;
        /* Create the task without using any dynamic memory allocation. */
        xHandle = xTaskCreateStatic(
                      vTaskCode,       /* Function that implements the task.
*/
                      "NAME",          /* Text name for the task. */
                      STACK_SIZE,      /* Number of indexes in the xStack
array. */
                      ( void * ) 1,    /* Parameter passed into the task. */
                      tskIDLE_PRIORITY,/* Priority at which the task is
created. */
                      xStack,          /* Array to use as the task's stack.
*/
                      &xTaskBuffer );  /* Variable to hold the task's data
structure. */

        /* puxStackBuffer and pxTaskBuffer were not NULL, so the task will
have
        been created, and xHandle will be the task's handle.  Use the handle
        to suspend the task. */
        vTaskSuspend( xHandle );
    }
```

vTaskDelete
[Task Creation]

task. h
void vTaskDelete(TaskHandle_t xTask);

INCLUDE_vTaskDelete must be defined as 1 for this function to be available.

Remove a task from the RTOS kernels management. The task being deleted will be removed from all ready, blocked, suspended and event lists.

NOTE: The idle task is responsible for freeing the RTOS kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of microcontroller processing time if your application makes any calls to vTaskDelete (). Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.

See the demo application file death. c for sample code that utilises vTaskDelete ().

Parameters:

xTask	The handle of the task to be deleted. Passing NULL will cause the calling task to be deleted.

Example usage:

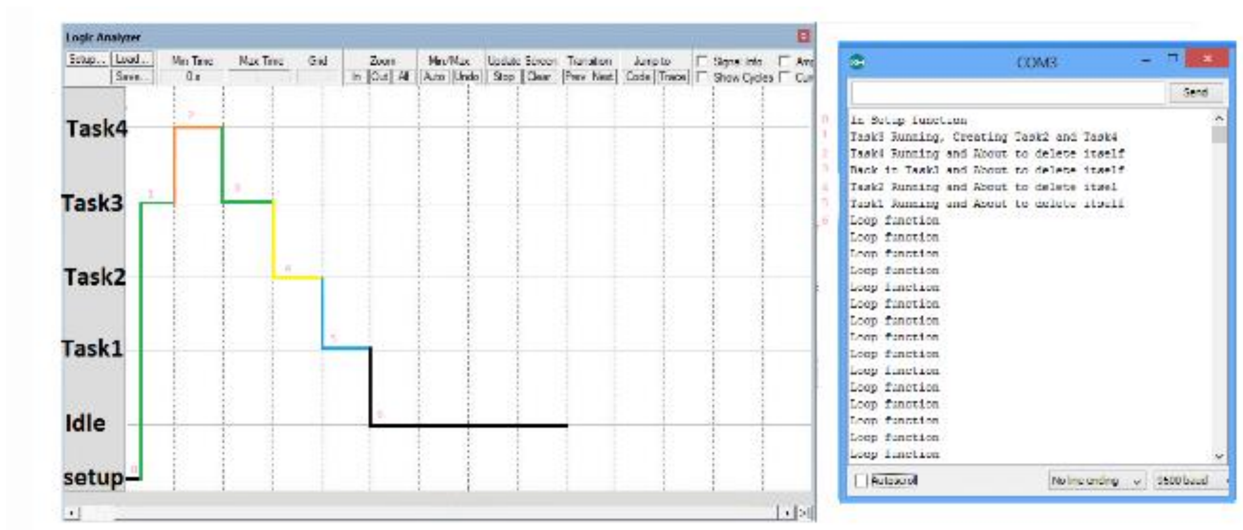
```
 void vOtherFunction( void )
 {
 TaskHandle_t xHandle = NULL;
     // Create the task, storing the handle.
     xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY,
&xHandle );
     // Use the handle to delete the task.
     if( xHandle != NULL )
     {
         vTaskDelete( xHandle );
     }
 }
```

Sample program in Arduino to create task

```
#include <Arduino_FreeRTOS.h>

TaskHandle_t TaskHandle_1;
TaskHandle_t TaskHandle_2;
TaskHandle_t TaskHandle_3;
TaskHandle_t TaskHandle_4;

void setup()
{

Serial.begin(9600);
Serial.println(F("In Setup function"));

/* Create two tasks with priorities 1 and 3. Capture the Task details to respective handlers */
xTaskCreate(MyTask1, "Task1", 100, NULL, 1, &TaskHandle_1);
xTaskCreate(MyTask3, "Task3", 120, NULL, 3, &TaskHandle_3);

}

void loop()
{
// Hooked to IDle task, it will run whenever CPU is idle
Serial.println(F("Loop function"));
delay(50);
}
```

```
/* Task1 with priority 1 */
static void MyTask1(void* pvParameters)
{
while(1)
{
Serial.println(F("Task1 Running and About to delete itself"));
vTaskDelete(TaskHandle_1); // Delete the task using the TaskHandle_1
}
}

/* Task2 with priority 2 */
static void MyTask2(void* pvParameters)
{
while(1)
{
Serial.println(F("Task2 Running and About to delete itsel"));
vTaskDelete(NULL); //Delete own task by passing NULL(TaskHandle_2 can also be used)
}
}

/* Task3 with priority 3 */
static void MyTask3(void* pvParameters)
{
while(1)
{
Serial.println(F("Task3 Running, Creating Task2 and Task4"));
xTaskCreate(MyTask2, "Task2", 50, NULL, 2, &TaskHandle_2);
xTaskCreate(MyTask4, "Task4", 100, NULL, 4, &TaskHandle_4);

Serial.println(F("Back in Task3 and About to delete itself"));
vTaskDelete(TaskHandle_3);
}
}

/* Task4 with priority 4 */
static void MyTask4(void* pvParameters)
{
while(1)
{
Serial.println(F("Task4 Running and About to delete itself"));
vTaskDelete(NULL); //Delete own task by passing NULL(TaskHandle_2 can also be used)
}
}
```

Output



```
In Setup function
Task3 Running, Creating Task2 and Task4
Task4 Running and About to delete itself
Back in Task3 and About to delete itself
Task2 Running and About to delete itsel
Task1 Running and About to delete itself
Loop function
Loop function
Loop function
Loop function
Loop function
Loop function
Loop function
Loop function
Loop function
Loop function
Loop function
Loop function
Loop function
Loop function
Loop function
```

0. Serial port is initialized and 2-Tasks are created with different priorities.

1. Cpu chooses Task3 out of the tasks(1,3,and idle) as it has higher priority. Now Task3 will create two more tasks with priority2,4. Now CPU has tasks(1,2,3,4,IDLE). Since Task4 has the highest priority it will preempt the Task3 and starts running.
2. Task4 will run for some time and deletes itself.

3. Now Tasks(1,2,3,IDLE) are available and Task3 will run again because for its higher priority. It will Run for some time and deletes itself.
4. Now Tasks(1,2,Idle) are remaining and Task2 will run and deletes itself once its job is done.
5. Out of Task(1,Idle), Task1 runs and it also deletes itself.
6. CPU is left out with the Idle task and it keeps running.

Semaphores

Binary resource semaphore

A binary resource semaphore is used to control sharing a single resource between tasks. Its internal counter can have only the values of 1 (available) and 0 (unavailable). A semaphore test passes if the count is 1, in which case, the current task is allowed to proceed. If the count is 0, the *semaphore test* fails and the current task is enqueued in the semaphore's task wait list, in priority order. When the semaphore is *signaled*, the first task (i.e. the longest-waiting, highest-priority task) is resumed, and the internal semaphore count remains 0. If no task is waiting, the count is increased to 1. The following is an example of using a binary resource semaphore:

```
    TCB_PTR t2a, t3a; // tasks
    SCB_PTR sbr; // binary resource semaphore

    void Init(void) {
       sbr = smx_SemCreate(RSRC, 1, "sbr");
       t2a = smx_TaskCreate(t2a_main, Pri2, 500, NO_FLAGS, "t2a"); //
500 byte stack
       t3a = smx_TaskCreate(t3a_main, Pri3, 500, NO_FLAGS, "t3a");
       smx_TaskStart(t2a);
    }

    void t2a_main(void) {
       if (smx_SemTest(sbr, TMO)) { // TMO = timeout in ticks
          smx_TaskStart(t3a);
          // use resource here
          smx_SemSignal(sbr);
       }
       else
          // deal with timeout or error
    }

    void t3a_main(void) {
       if (smx_SemTest(sbr, TMO)) {
          // use resource here
          smx_SemSignal(sbr);
```

```
      }
      else
         // deal with timeout or error
   }
```

In this example, semaphore sbr is created and its count is set to 1, making it a binary semaphore and as if it had already been signaled -- i.e., the resource is available. Task t2a (priority 2, task a) runs first and "gets" sbr. t2a then starts task t3a (priority 3, task a), which immediately preempts and tests sbr. Since sbr's count == 0, t3a is suspended on sbr, and t2a resumes. t2a uses the resource, then signals sbr, when it is done with the resource. This causes t3a to be resumed and to preempt t2a. t3a is now able to use the resource. When done, t3a signals sbr so that another task can use the resource and t3a stops. t2a resumes and stops.

In the case of a timeout or other error, SemTest() returns FALSE, meaning that the test did not pass and the resource is not free.

Multiple resource semaphore

The multiple resource semaphore is a generalization of the binary resource semaphore, intended to regulate access to multiple identical resources. It is commonly called a counting semaphore.This semaphore cannot be replaced by a mutex because the latter can only control access to one resource. The following is an example of using a multiple resource semaphore to control access to a block pool containing NUM blocks (for simplicity, timeouts and return value testing have been omitted -- this is NOT recommended in real code):

```
TCB_PTR t2a, t3a; // tasks
PCB_PTR blk_pool; // block pool
SCB_PTR sr; // resource semaphore
#define NUM 10
#define SIZE 100

void Init(void) {
   u8* p = (u8*)smx_HeapMalloc(NUM*SIZE);
   sb_BlockPoolCreate(p, blk_pool, NUM, SIZE, "blk pool");
   sr = smx_SemCreate(RSRC, NUM, "sr");
   t2a = smx_TaskCreate(t2a_main, Pri2, 500, NO_FLAGS, "t2a");
   t3a = smx_TaskCreate(t3a_main, Pri3, 500, NO_FLAGS, "t3a");
   smx_TaskStart(t2a);
}

void t2a_main(void) {
   u8* bp;
   smx_SemTest(sr, INF);
```

```
      bp = sb_BlockGet(blk_pool, 0));
      smx_TaskStart(t3a);
      // use bp to access block
      sb_BlockRel(blk_pool, bp, 0);
      smx_SemSignal(sr);
   }

   void t3a_main(void) {
      u8* bp;
      smx_SemTest(sr, INF);
      bp = sb_BlockGet(blk_pool, 0));
      // use bp to access block
      sb_BlockRel(blk_pool, bp, 0);
      smx_SemSignal(sr);
   }
```

This example is similar to the previous example, with small differences. The Init() function first creates a block pool of NUM blocks. It then creates sr, but because NUM is used instead of 1, a multiple resource semaphore is created, with a starting count of NUM. t2a tests sr and sr's counter is decremented to 9, which is greater than 0, so t2a is allowed to get a block. As before, t2a starts t3a, which immediately preempts. t3a tests sr and sr's counter is decremented to 8, so t3a is also allowed to get a block and use it. Eight more blocks can be obtained from blk_pool and used by the same or different tasks. However, the eleventh test of sr will suspend the testing task on sr. As shown, when t3a is done with its block, it releases the block back to blk_pool, then signals sr, thus allowing the first waiting task at sr to get the released block. Similarly for t2a.

If no task is waiting, sr's count is incremented by each signal. Thus the count always equals the number of available blocks. The maximum possible count is NUM. Signals after the count == NUM are ignored in order to protect resources if redundant signals should occur.

*******Example*****

The following example shows how to use a semaphore for general synchronization. Assume Task A should be activated when Task B has reached a certain point (Task B is the main program):

```
#include <stdio.h>
#include <Rtk32.h>

RTKSemaphore S;

void RTKAPI TaskA(void * P)
{
   printf("Task A: waiting on semaphore S\n");
   RTKWait(S);
   printf("Task A: continued\n");
}

int main(void)
{
   printf("\n");
   RTKernelInit(3);
   S = RTKCreateSemaphore(ST_COUNTING, 0, "Semaphore S");
   printf("Main  : creating task A\n");
   RTKRTLCreateThread(TaskA, 4, 0, 0, NULL, "Task A");
   printf("Main  : setting semaphore S\n");
   RTKSignal(S);
   printf("Main  : done.\n");
   return 0;
}
```

The semaphore is initialized with 0 events. Task A has a higher priority and thus is started immediately after RTKCreateThread. Its second statement is RTKWait(S). Since there are no events stored, Task A is blocked and the main task runs until the semaphore is signalled. When the main task executes the statement RTKSignal(S), Task A is activated immediately and retrieves the event. Both tasks then run to completion and the program terminates.