HuskEOS

Priority-based preemptive Real-Time Operating System

Written for HuskEOS version 2.x

August 2019

**** IN WORK ****

Developers:

Garrett Sculthorpe

Darren Cicala

Background

This project was started in February 2019 by Garrett Sculthorpe as an effort to learn more about task scheduling in embedded software design. The first implementation was a simple cooperative scheduler with some peripheral modules. After about two months, this was changed to a priority-based preemptive scheduler, and some typical RTOS modules were implemented shortly thereafter (queue, mailbox, semaphore, and flags). These modules made up what would be considered HuskEOS version 1.

Eventually the decision was made to add new functionality to the scheduler while also improving performance (namely reducing overhead time). This rewrite more closely followed a layered approach, allowing all RTOS modules to be fully portable to any microcontroller dependent only on the CPU_OS_Interface internal module. This allowed for a rewrite of the other modules as well, as the updated scheduler APIs were written to better support these modules and their requirements. Scheduler data structure handling was also abstracted into the list manager module, and other modules were changed to use the list manager module for their blocked task handling as well. A mutex module with priority inheritance was then added, and Darren Cicala joined to write the memory module.

As it currently stands, HuskEOS consists of seven public modules available to the user and two private modules for internal OS use, as described in this document. Each module supports optional task blocking for applicable API calls. Metrics can be found in the appendix. Any questions or comments regarding implementation or design, or any bugs to report, can be sent to gscultho@umich.edu and/or djcicala@umich.edu.

Github repository: https://github.com/gscultho/HuskEOS

Contents

1 Overview

HuskEOS is a Real-Time Operating System (RTOS) consisting of ten modules with eight of them for application use. The architecture is designed such that the RTOS will act as its own layer in a software project, and it has two internal sublayers. The first is the functional modules, being: scheduler, event flags, semaphore, mutex, mailbox, queue, and memory. It also includes a private module, list manager, that is used extensively by the scheduler for maintaining its task queues, as well as by the other modules for maintaining lists of blocked tasks on their resources.

The second internal sublayer is the CPU/OS interface module, which allows for the rest of the RTOS to be entirely abstracted from the hardware it is used on. This module supports nested critical sections, interrupt priority masking, the context switch, and routines for performing task initialization and system tick configuration. Any functionality that is needed from this layer at the application layer is mapped through functions or macros in the scheduler module, so CPU/OS interface is entirely “hidden” from the application developers. So, to port the RTOS to a microcontroller, only this layer needs to be re-written, consisting of only one C source file, one header file, and one ASM file.

This layered view of a potential system is then shown below:

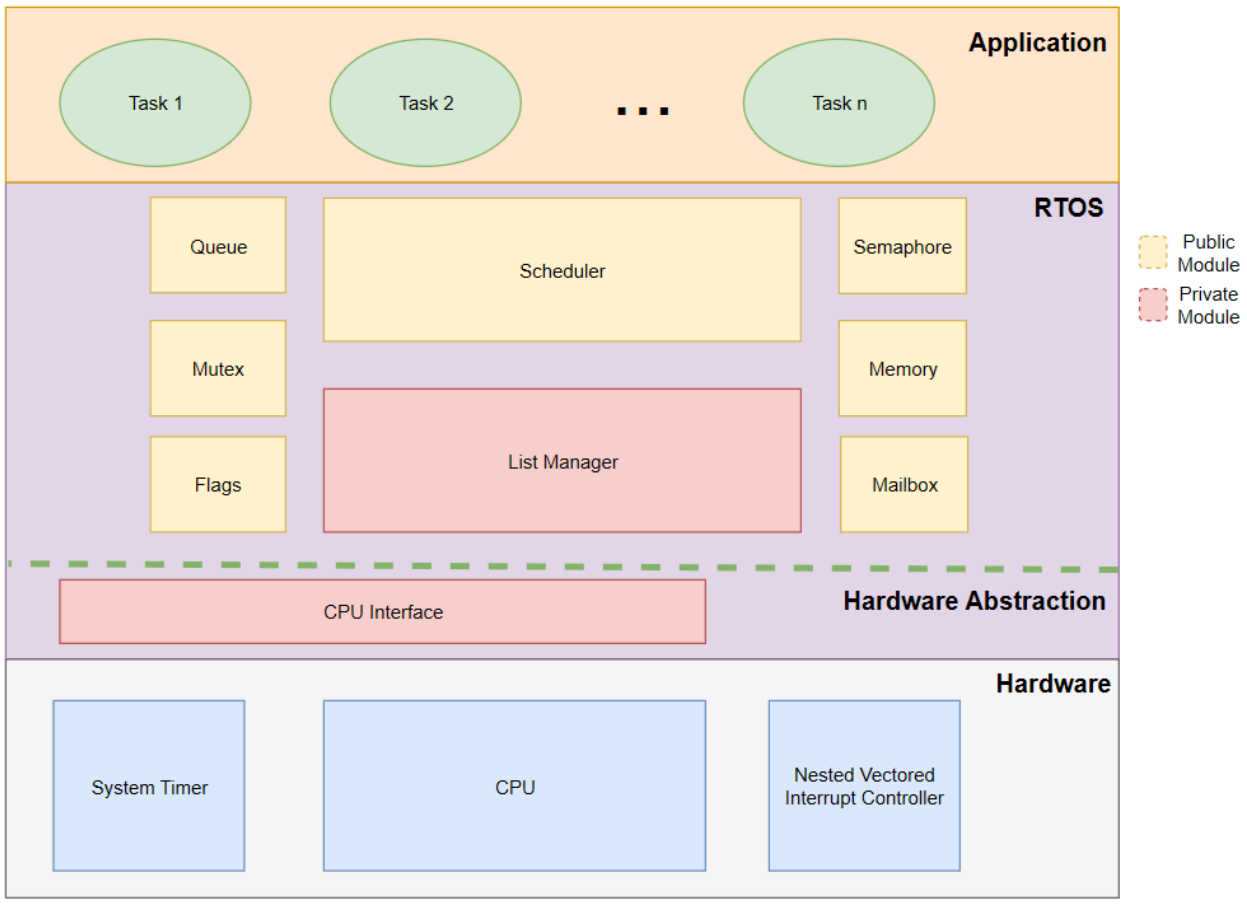

Figure 1.1: Layered architecture approach with HuskEOS.

It is also seen that other modules can easily be added to the system and even use the RTOS modules as well, such as additional hardware drivers. These would exist in parallel to the operating system.

Each module supports optional task blocking with timeout, configurable blocked list lengths, and other configurable options depending on the module.

2 Configuration and Startup

The goal of this section is to show how HuskEOS can be configured, and in what ways the RTOS needs to be initialized on startup.

2.1 Naming Conventions

Note the following variable, type, and function naming conventions commonly used below:

Variable Types:
- unsigned char: `U1`
- signed char: `S1`
- unsigned short (two bytes): `U2`
- signed short (two bytes): `S2`
- unsigned int (four bytes): `U4`
- signed int (four bytes): `S4`

Unsigned/signed long (eight bytes) are also defined as `U8`/`S8` but are not used in the current implementation.

Function Names:
*returnType*_OS*module*_*camelCaseName*(arg1, arg2,…)

Where *returnType* is the return variable type, *module* is either the full or abbreviated name of the module, and *camelCaseName* is the name of the function, typically describing its purpose, in camel case.

Variable Names:
type_scope_camelCaseName

Where *type* is the type of variable (`U1`, `U4`,…), *scope* is "t" for auto, "s" for static, "g" for global, and *camelCaseName* is the name of the variable. Additionally, if the name is for an array, pointer, and/or struct, then "a", "p", and/or "s" are typically appended to the end of the *scope.*

2.2 Configuration

Configuration options are to be changed in the header file rtos_cfg.h:

```
/***************************************************************************/
/*  Definitions                                                            */
/***************************************************************************/
#define RTOS_CONFIG_TRUE                              (1)
#define RTOS_CONFIG_FALSE                             (0)

/* Application */
#define RTOS_CONFIG_BG_TASK_STACK_SIZE                (50)                 /* Stack size for background task if enabled */
#define RTOS_CONFIG_CALC_TASK_CPU_LOAD                (RTOS_CONFIG_FALSE)  /* Can only be enabled if RTOS_CONFIG_BG_TASK and
                                                                              RTOS_CONFIG_ENABLE_BACKGROUND_IDLE_SLEEP enabled */
/* Scheduling */
#define RTOS_CONFIG_MAX_NUM_TASKS                     (6)                  /* This number of TCBs will be allocated at compile-
                                                                              time, plus any others used by OS */
                                                                           /* Available priorities are 0 - 0xEF with 0 being
                                                                              highest priority. */
#define RTOS_CONFIG_ENABLE_BACKGROUND_IDLE_SLEEP      (RTOS_CONFIG_TRUE)   /* Can only be used if RTOS_CONFIG_BG_TASK is enabled */
#define RTOS_CONFIG_ENABLE_STACK_OVERFLOW_DETECT      (RTOS_CONFIG_TRUE)   /* Can only be used if RTOS_CONFIG_BG_TASK is enabled */
#define RTOS_CONFIG_PRESLEEP_FUNC                     (RTOS_CONFIG_FALSE)  /* If enabled, hook function app_OSPreSleepFcn() can be
                                                                              defined in application. */
#define RTOS_CONFIG_POSTSLEEP_FUNC                    (RTOS_CONFIG_FALSE)  /* If enabled, hook function app_OSPostSleepFcn() can be
                                                                              defined in application. */

/* Mailbox */
#define RTOS_CFG_OS_MAILBOX_ENABLED                   (RTOS_CONFIG_TRUE)
#define RTOS_CFG_NUM_MAILBOX                          (3)                   /* Number of mailboxes available in run-time. */
#define RTOS_CFG_MBOX_DATA                           U4                     /* Data type for mailbox */

/* Message Queues */
#define RTOS_CFG_OS_QUEUE_ENABLED                     (RTOS_CONFIG_TRUE)
#define RTOS_CFG_NUM_FIFO                             (3)                   /* Number of FIFOs available in run-time. */
#define RTOS_CFG_MAX_NUM_BLOCKED_TASKS_FIFO           (3)                   /* Maximum number of tasks that can block on each FIFO.
                                                                            */
#define RTOS_CFG_BUFFER_DATA                          U4                    /* Type of data used in the buffers. */

/* Semaphores */
#define RTOS_CFG_OS_SEMAPHORE_ENABLED                 (RTOS_CONFIG_TRUE)
#define RTOS_CFG_NUM_SEMAPHORES                       (2)                   /* Number of semaphores available in run-time. */
#define RTOS_CFG_NUM_BLOCKED_TASKS_SEMA               (3)                   /* Maximum number of tasks that can block on each FIFO.
*/

/* Flags */
#define RTOS_CFG_OS_FLAGS_ENABLED                     (RTOS_CONFIG_TRUE)
#define RTOS_CFG_NUM_FLAG_OBJECTS                     (2)                   /* Number of flag objects available in run-time. */
#define RTOS_CFG_MAX_NUM_TASKS_PEND_FLAGS             (2)                   /* Maximum number of tasks that can pend on flags
                                                                               object. */

/* Mutex */
#define RTOS_CFG_OS_MUTEX_ENABLED                     (RTOS_CONFIG_TRUE)
#define RTOS_CFG_MAX_NUM_MUTEX                        (2)                   /* Number of mutexes available in run-time. */
#define RTOS_CFG_MAX_NUM_BLOCKED_TASKS_MUTEX          (2)                   /* Number of tasks that can block on each mutex. */

/* Memory */
#define RTOS_CFG_MAX_NUM_MEM_BLOCKS                   (16)
```

Starting from the top, `RTOS_CONFIG_BG_TASK_STACK_SIZE` exists so that the user can increase the background task stack if needed. This may be necessary due to enabling `RTOS_CONFIG_CALC_TASK_CPU_LOAD` or `RTOS_CONFIG_PRESLEEP_FUNC`. If CPU load calculation is not needed, for example, memory can be saved by decreasing the background task stack size. If `RTOS_CONFIG_PRESLEEP_FUNC` is enabled, the prototype for a hook function is provided for the application to define. This function is called immediately prior to the RTOS putting the CPU state to sleep when there are no active tasks. If this function call requires more stack memory, then the background stack would need to be increased. However, this is only applicable if both `RTOS_CONFIG_BG_TASK` and `RTOS_CONFIG_ENABLE_BACKGROUND_IDLE_SLEEP` are enabled. `RTOS_CONFIG_POSTSLEEP_FUNC` is defined with a similar functionality, but for after the CPU wakes back up, and is not dependent on the background task stack memory.

Stack overflow detection is also configurable, and if enabled, will put the CPU in a fault state if stack overflow occurs.

Each module that contains resources for application use has similar configurability options. First, the module must be enabled. Second, the total number of resources must be defined. Lastly, the maximum number of tasks that can pend on each resource at any given time must be defined as well. This is due to the RTOS not using heap memory, and so all memory used by the RTOS must be allocated at compile-time. If dynamic memory functionality is desired, then the memory module serves to mimic a heap while actually using static memory underneath. There are some differences between each module and its configurations, and so the section in this document detailing the use of the desired module should be read for a better understanding before use.

2.3 Startup

HuskEOS does not contain any low-level startup code as this varies depending on the platform. Startup code has been modified for the ARM Cortex-M4 from open source Texas Instruments startup code and is included in the repository. Essentially, startup code must do the following:

1: Declare system tick interrupt vector and place in memory.

2: Initialize stack pointer.

3: Copy DATA section from flash into RAM and initialize the values.

4: Jump to main (in application).

After jumping to main, the following sequence of events must occur (other application tasks can occur in between or before, but not after step 4):

1: Call `vd_OS_init(TIME_PERIOD)` where `TIME_PERIOD` is the number of milliseconds per system tick.

2: For each task to be created, create it by calling `u1_OSsch_createTask()`. Find API documentation in the scheduler module section of this document.

3: Initialize all RTOS resources to be used. Find API documentation in the corresponding section of this document for the desired resource.

4: Call `vd_OSsch_start()`. After this point program execution will not return to `main()` again.

3 Public Modules

This section includes some design information and API descriptions for each public module in the operating system.

3.1 Scheduler

The scheduler module consists of files sch.h, sch_internal_IF.h, and sch.c. The application should not reference sch_internal_IF.h, all necessary APIs and definitions for application are in sch.h. The scheduler acts as the main component of the kernel, as it handles the application task scheduling and state management, as well as performing periodic tasks for the kernel in the background (lowest priority).

3.1.1 Module Design

The scheduler has three internal data structures used to schedule tasks, manage their states, and provide task blocking functionality to other modules. Tasks can have three states: "ready", "sleeping", and "suspended". Sleep state has a timeout, while suspended means that the task is sleeping until explicitly woken up by the application.

The first data structure used is a doubly linked list of all tasks in the ready state. The scheduler uses the list manager module in order to create and maintain this list, using the `ListNode` structure found in listMgr_internal.h. This structure is seen below

```
typedef struct ListNode
{
  struct ListNode* nextNode;
  struct ListNode* previousNode;
  struct Sch_Task* TCB;
}
ListNode;
```

The entries are pointers to the next node in the list, the previous node in the list, and the task control block (TCB) of the task that the node references. This linked list is ordered by priority, and tasks are added to it in order of priority whenever they exit the sleep or suspended state (`SCH_TASK_FLAG_STS_SLEEP` and `SCH_TASK_FLAG_STS_SUSPENDED` respectively in sch.c). This list is pointed to by `node_s_p_headOfReadyList` in sch.c, a static pointer of type `ListNode`. The TCB structure is shown below, found in sch_internal_IF.h:

```
typedef struct Sch_Task
{
  OS_STACK*   stackPtr;         /* Task stack pointer must be first entry in
                                   struct. */
  U1          priority;         /* Task priority. */
  U1          taskID;           /* Task ID. Task can be referenced via this
                                   number. */
  U1          flags;            /* Status flags used for scheduling. */
  U4          sleepCntr;        /* Sleep counter. Unit is scheduler ticks. */
  void*       resource;         /* If task is blocked on a resource, its
                                   address is stored here. */
  U1          wakeReason;       /* Stores code for reason task was most
                                   recently woken up. */
#if(RTOS_CONFIG_ENABLE_STACK_OVERFLOW_DETECT == RTOS_CONFIG_TRUE)
  OS_STACK*   topOfStack;       /* Pointer to stack watermark. Used to detect
                                   stack overflow. */
#endif
}
Sch_Task;
```

The second data structure is similarly a doubly linked list of all tasks in the sleep or suspended state. These tasks are ordered in no specific order other than the sleeping tasks being placed at the front of the linked list, while the suspended tasks are placed in the back. This was done so that when parsing this list and checking the task sleep counters, the parsing can stop once it hits a suspended task. This list is pointed to by `node_s_p_headOfWaitList`, also a static pointer of type `ListNode` in sch.c. When a task is moved between these two lists, the entire `ListNode` structure (via changing the next/previous pointers) is moved to the list. So, each task has one `ListNode` structure in memory that gets moved between the scheduler lists.

The third data structure is used to map taskIDs, which are used to identify tasks at the application layer, to their respective `ListNode` structures. This is simply an array of pointers of type `ListNode`, and the array is called `Node_s_ap_mapTaskIDToTCB[]` in sch.c. For example, `Node_s_ap_mapTaskIDToTCB[2]` maps to the application task's `ListNode` structure in memory whose taskID is set to 2.

All of the above structures are internal and should not be referenced by the application. This information is included for purposes of understanding the module only. `Node_s_ap_mapTaskIDToTCB[]` is left as a global array without the static qualifier so that read-only macros can be put in sch_internal_IF.h for fast taskID mapping for internal modules, but this should not be used by the application.

Below is an illustration of how these structures interact with one another:

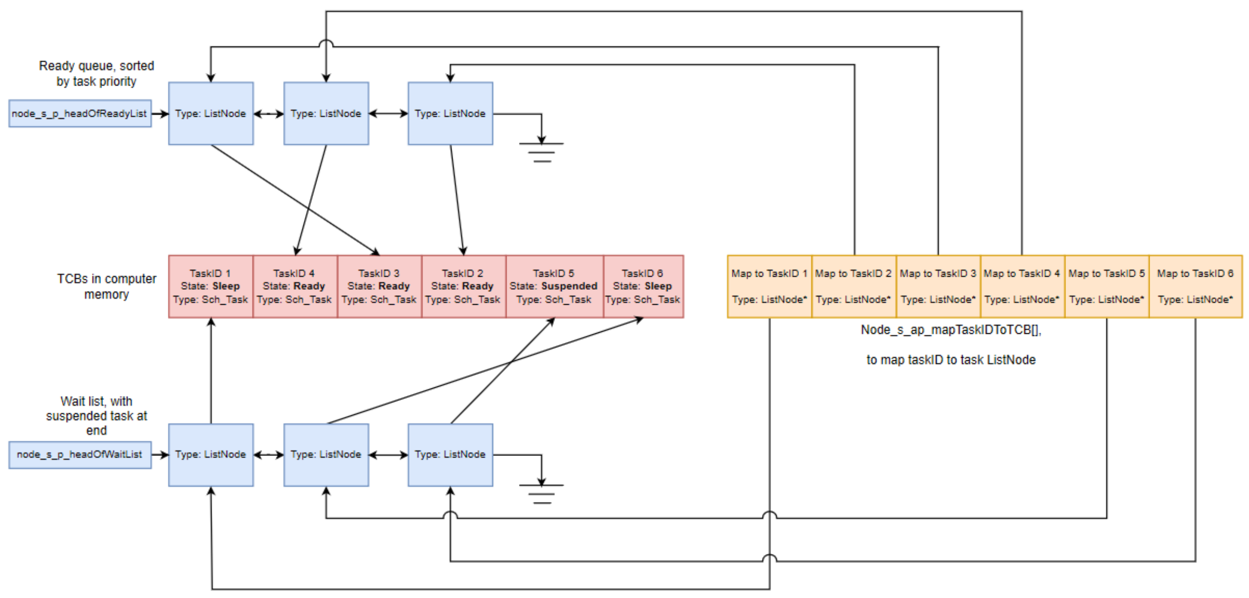
Ready queue, sorted by task priority
node_s_p_headOfReadyList
Type: ListNode
Type: ListNode
Type: ListNode
TCBs in computer memory
TaskID 1 State: Sleep Type: Sch_Task
TaskID 4 State: Ready Type: Sch_Task
TaskID 3 State: Ready Type: Sch_Task
TaskID 2 State: Ready Type: Sch_Task
TaskID 5 State: Suspended Type: Sch_Task
TaskID 6 State: Sleep Type: Sch_Task
Map to TaskID 1 Type: ListNode*
Map to TaskID 2 Type: ListNode*
Map to TaskID 3 Type: ListNode*
Map to TaskID 4 Type: ListNode*
Map to TaskID 5 Type: ListNode*
Map to TaskID 6 Type: ListNode*
Node_s_ap_mapTaskIDToTCB[],
to map taskID to task ListNode
Wait list, with suspended task at end
node_s_p_headOfWaitList
Type: ListNode
Type: ListNode
Type: ListNode

Figure 3.1: Scheduler data structures assuming six tasks exist.

Referencing the diagram above, the tasks exist in this case such that the priority of task with ID 4 > priority of taskID 3 > priority of taskID 2, etc. TaskID is not associated with priority in any way, as seen in the documentation for `u1_OSsch_createTask()`. It is also seen that the tasks in the sleep state are at the front of the wait list, while the suspended task is in the back. Lastly, each task's `ListNode` can be quickly dereferenced using the taskID and `Node_s_ap_mapTaskIDToTCB[]`. As previously mentioned, this structure exists due to the application referencing the tasks through this taskID number, requiring a lookup table for fast access.

3.1.2 Module Public Functions and Macros

The following functions and macros are declared in sch.h and can be accessed via this header file.

3.1.2.1 Macros *OS_SCH_ENTER_CRITICAL* and *OS_SCH_EXIT_CRITICAL*

Definition:

```
#define OS_SCH_ENTER_CRITICAL(void)
#define OS_SCH_EXIT_CRITICAL(void)
```

Purpose:

Enter/exit a critical section by disabling/enabling interrupts. Nesting is supported. Implementation details are dependent on CPU/OS Interface module. Conceptually, though, nesting is implemented by atomically incrementing or decrementing a counter variable and using this to track the number of sequential enter/exit function calls.

Arguments:

N/A

Return:

N/A

Example:

```
void func1(void);

int main()
{
  OS_SCH_ENTER_CRITICAL();

  ...

  func1();

  /* Interrupts still disabled due to CS nesting. */
  ...

  OS_SCH_EXIT_CRITICAL(); /* Interrupts enabled at this point. */

}

void func1(void)
{
  OS_SCH_ENTER_CRITICAL();

  ...

  OS_SCH_EXIT_CRITICAL();
}
```

3.1.2.2 Macros *u1_OSsch_maskInterrupts* and *vd_OSsch_unmaskInterrupts*

Definition:

```
#define u1_OSsch_maskInterrupts(c)
#define vd_OSsch_unmaskInterrupts(c)
```

Purpose:

Used together to mask and unmask system tick interrupts.

Arguments:

Void and previous interrupt mask. This previous mask is returned from `u1_OSsch_maskInterrupts` in a single byte.

Return:

Previous interrupt mask (U1) and void.

Example:

```
int main()
{
  U1 u1_t_prevMask;

  u1_t_prevMask = u1_OSsch_maskInterrupts();

  ...

  vd_OSsch_unmaskInterrupts(u1_t_prevMask);
}
```

3.1.2.3 Function *vd_OS_init*

Definition:

```
void vd_OS_init(U4 numMsPeriod);
```

Purpose:

Used to initialize operating system on startup.

Arguments:

`U4 numMsPeriod`: Sets scheduler tick rate in milliseconds. Maximum value is `0xFFFFFFFF`.

Return:

N/A

Example:

```
int main()
{
  vd_OS_init(2); /* Set tick period to 2ms per tick. */

  /* Initialize OS resources. */

  ...

  /* Initialize application. */

  ...

  /* Start OS. */

}
```

3.1.2.4 Function *u1_OSsch_createTask*

Definition:

```
U1 u1_OSsch_createTask(void (*newTaskFcn)(void), void* sp, U4
sizeOfStack, U1 priority, U1 taskID);
```

Purpose:

Create a single application task.

Arguments:

`void (*newTaskFcn)(void)`: Function pointer to task definition.
`void* sp`: Pointer to bottom of task stack.
`U4 sizeOfStack`: Size of task stack, unit is OS_STACK.
`U1 priority`: Priority of task (zero is highest priority, 0xEF is lowest).
`U1 taskID`: ID for task. Two tasks cannot share the same task ID. 0x01 – 0xFF are available.

Return:

`SCH_TASK_CREATE_SUCCESS` or `SCH_TASK_CREATE_DENIED`

Example:

```
OS_STACK task1Stack[100];


static void task1Fcn(void)
{
  while(1)
  {
    ...
  }
}

int main()
{
  U1 u1_t_taskCreateRtn;
  vd_OS_init(2); /* Set tick period to 2ms per tick. */

  u1_t_taskCreateRtn = u1_OSsch_createTask(&task1Fcn, &task1Stack[99],
  100, 5, 1);

  ...

  /* Initialize application. */

  ...

  /* Start OS. */

}
```

3.1.2.5 Function *vd_OSsch_start*

Definition:

```
void vd_OSsch_start(void);
```

Purpose:

Hand execution control over to the operating system.

Arguments:

N/A

Return:

N/A

Example:

```
int main()
{
  vd_OS_init(2); /* Set tick period to 2ms per tick. */

  /* Create tasks. */

  ...

  /* Initialize application. */

  ...

  vd_OSsch_start(void);
}
```

3.1.2.6 Function *u1_OSsch_interruptEnter* and *vd_OSsch_interruptExit*

Definition:

```
U1 u1_OSsch_interruptEnter(void);
void vd_OSsch_interruptExit(U1 prioMaskReset);
```

Purpose:

These functions must be called at the start/end of any ISRs external to the OS.

Arguments:

Void and `U1 prioMaskReset.` Argument prioMaskReset is returned from `u1_OSsch_interruptEnter.`

Return:

Current interrupt priority mask (U1) and void.

Example:

```
void ISR_1(void)
{
  U1 u1_t_prevMask;

  u1_t_prevMask = u1_OSsch_interruptEnter();

  ...

  vd_OSsch_interruptExit(void);
}
```

3.1.2.7 Function *u1_OSsch_g_numTasks*

Definition:

```
U1 u1_OSsch_g_numTasks(void);
```

Purpose:

Return the number of tasks that have been created.

Arguments:

N/A

Return:

Number of tasks that have been created (U1).

Example:

```
void taskN(void)
{
  U1 u1_t_numTasks;

  ...
```

```
    while(1)
    {
      u1_t_numTasks = u1_OSsch_g_numTasks();

      ...
    }
  }
```

3.1.2.8 Function *u4_OSsch_getCurrentTickPeriodMs*

Definition:

```
U4 u4_OSsch_getCurrentTickPeriodMs(void);
```

Purpose:

Return the number of milliseconds in one scheduler tick.

Arguments:

N/A

Return:

Number of milliseconds per tick (U4).

Example:

```
void taskN(void)
{
  U4 U4_t_tickPeriodMs;

  ...

  while(1)
  {
    U4_t_tickPeriodMs = u4_OSsch_getCurrentTickPeriodMs();

    ...
  }
}
```

3.1.2.9 Function *u1_OSsch_getReasonForWakeup*

Definition:

```
U1 u1_OSsch_getReasonForWakeup(void);
```

Purpose:

Get code for reason that task was last woken up. Assumes that task making the function call is the task in question.

Arguments:

N/A

Return:

```
SCH_TASK_WAKEUP_SLEEP_TIMEOUT          OR
SCH_TASK_NO_WAKEUP_SINCE_LAST_CHECK    OR
SCH_TASK_WAKEUP_MBOX_READY             OR
SCH_TASK_WAKEUP_QUEUE_READY            OR
SCH_TASK_WAKEUP_SEMA_READY             OR
SCH_TASK_WAKEUP_FLAGS_EVENT            OR
SCH_TASK_WAKEUP_MUTEX_READY            OR
```

All `U1`.

Example:

```
void taskN(void)
{
  U1 u1_t_wakeupReason;

  ...

  while(1)
  {
    u1_t_wakeupReason = u1_OSsch_getReasonForWakeup();

    ...
  }
}
```

3.1.2.10 Function *u4_OSsch_getTicks*

Definition:

```
U4 u4_OSsch_getTicks(void);
```

Purpose:

Get number of ticks from the scheduler's internal running counter. Overflows at 0xFFFFFFFF to zero. This can be used for time-keeping at the application layer.

Arguments:

N/A

Return:

Number of scheduler ticks (U4).

Example:

```
void taskN(void)
{
  U4 u4_t_timeStamp;
```

```
    ...

    while(1)
    {
      u4_t_timeStamp = u4_OSsch_getTicks();

      ...
    }
  }
```

3.1.2.11 Function *u1_OSsch_getCurrentTaskID*

Definition:

```
U1 u1_OSsch_getCurrentTaskID(void);
```

Purpose:

Get the task ID for the currently executing task. Can be useful in an ISR, for example. The task ID is set by the application when the task is created.

Arguments:

N/A

Return:

ID number of currently executing task (U1).

Example:

```
void taskN(void)
{
  U4 u4_t_taskID;

  ...

  while(1)
  {
    u4_t_taskID = u1_OSsch_getCurrentTaskID();

    ...
  }
}
```

3.1.2.12 Function *u1_OSsch_getCurrentTaskPrio*

Definition:

```
U1 u1_OSsch_getCurrentTaskPrio(void);
```

Purpose:

Get the priority for the currently executing task. This API was created for the Mutex module but is accessible by the application as well.

Arguments:

N/A

Return:

Priority of currently executing task (U1).

Example:

```
void taskN(void)
{
  U1 u1_t_taskPrio;

  ...

  while(1)
  {
    u1_t_taskPrio = u1_OSsch_getCurrentTaskPrio();

    ...
  }
}
```

3.1.2.13 Function *u1_OSsch_getCPULoad*

Definition:

```
U1 u1_OSsch_getCPULoad(void);
```

Purpose:

Get the CPU load. This value is averaged over 100 system ticks, so calling the function more than once per 100 ticks will yield the same result. Only available if `RTOS_CONFIG_CALC_TASK_CPU_LOAD` is configured.

Arguments:

N/A

Return:

CPU load as a percentage out of 100 (U1).

Example:

```
void taskN(void)
{
  U1 u1_t_CPULoad;

  ...
```

```
    while(1)
    {
      u1_t_CPULoad = u1_OSsch_getCPULoad();

      ...
    }
  }
```

3.1.2.14 Function *vd_OSsch_setNewTickPeriod*

Definition:

```
  void vd_OSsch_setNewTickPeriod(U4 numMsReload);
```

Purpose:

Set new tick period for system tick. Unit is milliseconds.

Arguments:

`u4_t_newTickPeriod`: Tick period in milliseconds.

Return:

N/A

Example:

```
  void taskN(void)
  {
    U4 u4_t_newTickPeriod;

    u4_t_newTickPeriod = 2; /* 2ms period */
    ...

    while(1)
    {
      vd_OSsch_setNewTickPeriod(u4_t_newTickPeriod);

      ...
    }
  }
```

3.1.2.15 Function *vd_OSsch_taskSleep*

Definition:

```
  void vd_OSsch_taskSleep(U4 period);
```

Purpose:

Puts the task that calls this function to sleep for `period` number of system ticks. The task can also be woken up before this time by another task calling `vd_OSsch_taskWake`.

Arguments:

`period`: Number of system ticks for task to sleep.

Return:

N/A

Example:

```
void taskN(void)
{
  U4 u4_t_sleepPeriod;

  u4_t_sleepPeriod = 2; /* 2 tick period */

  ...

  while(1)
  {
    ...

    vd_OSsch_taskSleep(u4_t_sleepPeriod);
  }
}
```

3.1.2.16 Function *vd_OSsch_taskSuspend*

Definition:

```
void vd_OSsch_taskSuspend(U1 taskIndex);
```

Purpose:

Puts the task specified by `taskIndex` into the suspended state. This state has the task sleep indefinitely. The task can be woken up by another task calling `vd_OSsch_taskWake`.

Arguments:

`taskIndex`: TaskID of task to be put into suspended state (can be calling task, or any other task).

Return:

N/A

Example:

```
void taskN(void)
{
  U1 u1_t_taskID;

  u1_t_taskID = u1_OSsch_getCurrentTaskID();
  ...
  while(1)
  {
    ...

    vd_OSsch_taskSuspend(u1_t_taskID);
  }
}
```

3.1.2.17 Function *vd_OSsch_suspendScheduler*

Definition:

```
void vd_OSsch_suspendScheduler(void);
```

Purpose:

Turns off system ticks. They would need to be restarted by calling `vd_OSsch_setNewTickPeriod`().

Arguments:

N/A

Return:

N/A

Example:

```
void taskN(void)
{
  U1 u1_t_tickPerRestart;

  u1_t_tickPerRestart = 2; /* 2ms period for system tick */
  ...

  while(1)
  {
    vd_OSsch_suspendScheduler();

    ...

    vd_OSsch_setNewTickPeriod(u1_t_tickPerRestart);
  }
}
```

3.1.2.18 Functions *app_OSPreSleepFcn* and *app_OSPostSleepFcn*

These functions are declared in sch.h if `RTOS_CONFIG_PRESLEEP_FUNC` and `RTOS_CONFIG_POSTSLEEP_FUNC` are defined respectively. They are hook functions used to add application functionality before and/or after the CPU is put to sleep when no tasks are ready to run. `RTOS_CONFIG_ENABLE_BACKGROUND_IDLE_SLEEP` must also be configured for this functionality.

3.1.3 Module Private Functions and Macros

The following functions and macros are declared in sch_internal_IF.h and can be accessed via this header file. They are intended for internal OS use only.

3.1.3.1 Function *vd_OSsch_setReasonForWakeup*

Definition:

```
void vd_OSsch_setReasonForWakeup(U1 reason, struct Sch_Task*
wakeupTaskTCB);
```

Purpose:

When a task is blocked on a resource, like a semaphore for example, the semaphore keeps an internal record of this task being blocked on it. If the semaphore becomes available, it will then wake up the highest priority task waiting on it, and use this API to notify the scheduler of the reason that this task was woken up. This information can then be retrieved by the application by calling `u1_OSsch_getReasonForWakeup`().

Arguments:

`U1 reason:` Numeric code. See `u1_OSsch_getReasonForWakeup().`

`struct Sch_Task* wakeupTaskTCB`: Address of task TCB. This information is available to internal modules to optimize interfaces with scheduler with respect to execution speed. When an internal module stores this data, it must not be dereferenced and modified by the internal module.

Return:

N/A

Example:

```
static void vd_OSsema_unblockTask(OSSemaphore* semaphore)
{
  ListNode* node_t_p_highPrioTask;

  /* Remove highest priority task */
  node_t_p_highPrioTask = node_list_removeFirstNode(&(semaphore->
                                                     blockedListHead));

  /*  Notify scheduler the reason that task is going to be woken. */
  vd_OSsch_setReasonForWakeup((U1)SCH_TASK_WAKEUP_SEMA_READY,
                                  node_t_p_highPrioTask->TCB);

  /* Notify scheduler to change task state. If woken task is higher
     priority than running task, context switch will occur after
     critical section. */
  vd_OSsch_taskWake(node_t_p_highPrioTask->TCB->taskID);
```

```
    /* Clear TCB pointer. This frees this node for future use. */
    node_t_p_highPrioTask->TCB = SEMA_NULL_PTR;
  }
```

3.1.3.2 Function *vd_OSsch_setReasonForSleep*

Definition:

```
void vd_OSsch_setReasonForSleep(void* taskSleepResource, U1
resourceType);
```

Purpose:

If a task attempts to acquire a resource that supports blocking, the internal resource module will use this API to notify the scheduler that the task is going to sleep on this resource (numeric codes same as reasons for wakeup above). This does not put the task into the sleep state.

Arguments:

`void* taskSleepResource:` Memory address of OS resource task is blocking on.
`U1 resourceType`: Each resource type in the OS must have a numeric code defined in sch_internal_IF.h.

Return:

N/A

Example:

```
U1 u1_OSsema_wait(OSSemaphore* semaphore, U4 blockPeriod)
{
  U1 u1_t_returnSts;

  u1_t_returnSts = (U1)SEMA_SEMAPHORE_TAKEN;

  OS_SCH_ENTER_CRITICAL();

  /* Check if available */
  if(semaphore->sema == (U1)ZERO)
  {
    /* If non-blocking function call, exit critical section and return
       immediately */
    if(blockPeriod == (U4)SEMA_NO_BLOCK)
    {
      OS_SCH_EXIT_CRITICAL();
    }
    /* Else block task */
    else
    {
      /* Add task to resource blocked list */
      vd_OSsema_blockTask(semaphore);
      /* Tell scheduler the reason for task block state */
      vd_OSsch_setReasonForSleep(semaphore,
                                 (U1)SCH_TASK_SLEEP_RESOURCE_SEMA);

      /* Set sleep timer and change task state */
      vd_OSsch_taskSleep(blockPeriod);

      OS_SCH_EXIT_CRITICAL();

    ...
}
```

3.1.3.3 Function *u1_OSsch_setNewPriority*

Definition:

```
U1 u1_OSsch_setNewPriority(struct Sch_Task* tcb, U1 newPriority);
```

Purpose:

This API currently exists to support priority inheritance in the mutex module. Steps must be taken to ensure that no two tasks in the ready state share the same priority at any time.

Arguments:

`struct Sch_Task* tcb`: Memory address of OS resource task is blocking on.
`U1 newPriority`: Priority that is to be inherited by task, or task's true priority that is being restored after inheritance.

Return:

Previously set priority of task (U1).

Example:

```
static void vd_OSmutex_blockTask(OSMutex* mutex)
{
  U1 u1_t_priorityOriginal;

  u1_t_returnSts = (U1)SEMA_SEMAPHORE_TAKEN;

  OS_SCH_ENTER_CRITICAL();

  ...

  /* Notify scheduler of change. */
  u1_t_priorityOriginal = u1_OSsch_setNewPriority(mutex->
                              priority.mutexHolder,
                              mutex>blockedTaskList.blockedListHead->
                              TCB->priority);
  ...

  OS_SCH_EXIT_CRITICAL();

  ...
}
```

3.2 Flags

The flags module consists of files flags.h, flags_internal_IF.h, and flags.c. The application should not reference flags_internal_IF.h, all necessary APIs and definitions for application are in flags.h. Flags objects allow for signaling between different tasks. A task can pend on an event or set of events for a flags object and sleep until the event(s) occur(s), or a task can set/clear flags to be polled by any number of other tasks. The flags module has an internal interface with the scheduler through sch.h and sch_internal_IF.h.

3.2.1 Module Design

Any flags object is internally stored as the following structure:

```
typedef struct FlagsObj
{
  /* Object containing 8 flags. */
  U1 flags;

  /* Memory allocated for storing pending task info. */
  TasksPending  pendingList[FLAGS_MAX_NUM_TASKS_PENDING];
}
FlagsObj;
```

Member `flags` is a single byte storing the eight available binary flags. Member `TasksPending` is used to store data for any tasks that are pending on this flags object. Its structure is shown below:

```
typedef struct TasksPending
{
  U1                event;         /* Event that task is pending on. */
  struct Sch_Task* tcb;            /* Pointer to pending task TCB. */
  U1                eventPendType; /* Type of pend (exact match or just one
                                      flag set). */
}
TasksPending;
```

Member `event` contains the flag combination or set of flags that will wake the pending task up. There is also the address of the pending task TCB stored as reference to the task, and a binary member to indicate whether the task is pending on the exact flags in `event`, or rather any of the flags specified by `event`.

During startup the memory for each flags object needs to be allocated and initialized. See the API descriptions below for clarification. The maximum number of flags objects that can be created is defined by `RTOS_CFG_NUM_FLAG_OBJECTS` in rtos_cfg.h.

3.2.2 Module Public Functions

The following functions and macros are declared in flags_internal_IF.h and can be accessed via this header file.

3.2.2.1 Function *u1_OSflags_init*

Definition:

```
U1 u1_OSflags_init(OSFlagsObj** flags, U1 flagInitValues);
```

Purpose:

This function is used to initialize a flags object during the application initialization. The memory for the object must be allocated by the application by creating a pointer to type `OSFlagsObj`. This pointer is passed to the initialization routine and then used to reference the flags object during runtime.

Arguments:

`OSFlagsObj** flags:` Pass pointer to flags object by reference (pass its address).

`U1 flagInitValues`: Initial values of flags. Each flags object is a byte with each flag being one bit.

Return:

`FLAGS_NO_OBJ_AVAILABLE` OR
`FLAGS_INIT_SUCCESS`

All `U1`.

Example:

```
OSFlagsObj* Flags1Ptr;

...

int main()
{
  U1 u1_t_flagsInitSts;

  vd_OS_init(2); /* Set tick period to 2ms per tick. */

  ...

  u1_t_flagsInitSts = u1_OSflags_init(&Flags1Ptr, 0xF4); /* Initialize
                                                            flags to
                                                          0b11110100 */
  ...

  /* Create tasks. */

  ...

  /* Initialize application. */
```

```
    ...

    vd_OSsch_start(void);
  }
```

3.2.2.2 Function *u1_OSflags_postFlags*

Definition:

```
U1 u1_OSflags_postFlags(OSFlagsObj* flags, U1 flagMask, U1 set_clear);
```

Purpose:

This function is used to set or clear individual flags or groups of flags in a flags object. If any tasks are pending on an event posted by this function call, those tasks will be woken up. If a higher priority task is woken up by this function call, the task that calls this function will be preempted before this function returns.

Arguments:

`OSFlagsObj* flags:` Pointer to flags object.

`U1 flagMask`: Bitmask of flags to set/clear.

`U1 set_clear`: Either `FLAGS_WRITE_SET` (performs OR operation with `flagMask`) OR
`FLAGS_WRITE_CLEAR` (performs BITCLEAR operation).

Return:

`FLAGS_WRITE_COMMAND_INVALID` OR
`FLAGS_WRITE_SUCCESS`

All `U1`.

Example:

```
void taskN(void)
{
  U1 u1_t_flagsPostRtn;

  ...

  while(1)
  {
    ...

    /* Set flags with mask 0b11000111. */
    u1_t_flagsPostRtn = u1_OSflags_postFlags(Flags1Ptr, 0xC7,
                                             FLAGS_WRITE_SET);
    ...
  }
}
```

3.2.2.3 Function *u1_OSflags_pendOnFlags*

Definition:

```
U1 u1_OSflags_pendOnFlags(OSFlagsObj* flags, U1 eventMask, U4 timeOut,
                                                   U1 eventType);
```

Purpose:

This function is used to set a task to pend on an event for a group of flags. The event can be set as either all of a set of flags being set HIGH, or one of a set of flags being set HIGH. The set is determined by argument `eventMask`. The task can also retrieve the flags values as they were at the time that the task was woken up by calling `u1_OSsch_getReasonForWakeup()`. If no event occurs, the task will time out and wake up after `timeOut` number of system ticks. The maximum number of tasks that can pend on one flags object is defined by `RTOS_CFG_MAX_NUM_TASKS_PEND_FLAGS` in rtos_cfg.h.

Arguments:

`OSFlagsObj* flags`: Pointer to flags object.

`U1 eventMask`: Bitmask of flags to set/clear.

`U1 eventMask`: Timeout period measured in system ticks.

`U1 eventType`: Either `FLAGS_EVENT_ANY` (any flags in `eventMask`) OR
`FLAGS_EVENT_EXACT` (flags values match `eventMask` exactly).

Return:

```
FLAGS_PEND_LIST_FULL      OR
FLAGS_PEND_SUCCESS
```

All `U1`.

Example:

```
void taskN(void)
{
  U1 u1_t_flagsEvent;
  U1 u1_t_flagsPendRtn;
  U1 u1_t_flagsPendTimeout;
  U1 u1_t_flagsWakeupEvent;

  /* Event mask is 4 most significant bits HIGH. */
  u1_t_flagsEvent = 0xF0;

  /* Task pend will time out after 5 system ticks. */
  u1_t_flagsPendTimeout = 5;
  ...

  while(1)
  {
    ...

    /* Set pend type to any of the four flags. */
    u1_t_flagsPendRtn = u1_OSflags_pendOnFlags(Flags1Ptr, 0xC7,
                                               u1_t_flagsPendTimeout,
                                                 FLAGS_EVENT_ANY);

    /* Get flags at time of event. */
    u1_t_flagsWakeupEvent = u1_OSsch_getReasonForWakeup();

    ...
  }
}
```

3.2.2.4 Function *vd_OSflags_reset*

Definition:

```
void vd_OSflags_reset(OSFlagsObj* flags);
```

Purpose:

This function is used to reset the flags object to an initial state. All pending tasks are woken up, and the flags are cleared to zero.

Arguments:

`OSFlagsObj* flags`: Pointer to flags object.

Return:

N/A

Example:

```
void taskN(void)
{
  while(1)
  {
    ...

    /* Reset flags object and wake all pending tasks. */
    vd_OSflags_reset(flags1Ptr);

    ...
  }
}
```

3.2.2.5 Function *vd_OSflags_clearAll*

Definition:

```
void vd_OSflags_clearAll(OSFlagsObj* flags);
```

Purpose:

This function is used to clear all of the flags to zero, but will only wake up tasks that are pending on an event corresponding with this action.

Arguments:

`OSFlagsObj* flags:` Pointer to flags object.

Return:

N/A

Example:

```
void taskN(void)
{
  while(1)
  {
    ...

    /* Reset flag values. */
    vd_OSflags_clearAll(flags1Ptr);

    ...
  }
}
```

3.2.2.6 Function *u1_OSflags_checkFlags*

Definition:

```
U1 u1_OSflags_checkFlags(OSFlagsObj* flags);
```

Purpose:

This function is used to poll the flag values. The flags will not be modified.

Arguments:

`OSFlagsObj* flags:` Pointer to flags object.

Return:

Values of flags (U1).

Example:

```
void taskN(void)
{
  U1 u1_t_flagSts;

  while(1)
  {
    ...

    /* Reset flags object and wake all pending tasks. */
    u1_t_flagSts = u1_OSflags_checkFlags(flags1Ptr);

    ...
  }
}
```

3.2.3 Module Private Functions

3.2.3.1 Function *vd_OSflags_pendTimeout*

Definition:

```
void vd_OSflags_pendTimeout(struct FlagsObj* flags, struct Sch_Task*
                                                          pendingTCB);
```

Purpose:

This function exists as a callback for the scheduler. If a task times out while pending on a flags object, the scheduler calls this function. The flags module is then responsible for removing the task from its list of pending tasks and updating internal data as needed.

Arguments:

`FlagsObj* flags:` Pointer to flags object.

`struct Sch_Task* pendingTCB:` Pointer to task TCB. The pending task is stored by this value.

Return:

N/A

Example:

See sch.c.

3.3 Mailbox

The mailbox module consists of files mailbox.h, mbox_internal_IF.h, and mailbox.c. The application should not reference mbox_internal_IF.h, all necessary APIs and definitions for application are in mailbox.h.

Mailboxes allow for passing single pieces of data between two tasks. The sending task can put one piece of data into the mailbox and the receiving task must take it before more data can be put into the mailbox. The mailbox is designed to be used between two tasks such that one task acts only as the sender and the other task only as the receiver, but with some extra effort at the application layer the application can easily use the APIs such that both tasks send and receive. In the cases where the sender tries to put data into a mailbox with data already in it, or where the receiving task tries to get mail from an empty mailbox, task blocking with configurable timeout is supported. One task can block on a mailbox at a time as this is all that is required if the mailbox is only used between two tasks. The mailbox module has an internal interface with the scheduler through sch.h and sch_internal_IF.h.

3.3.1 Module Design

The mailbox module must be enabled by setting the definition of `RTOS_CFG_OS_MAILBOX_ENABLED` to 1 in rtos_cfh.h. Each mailbox has the following structure:

```
typedef struct Mailbox
{
  MAIL mail;              /* Holds data. */
  U1   blockedTaskID;     /* If a task is blocked on mailbox, its ID is stored
                             here. */
}
Mailbox;
```

Variable type `MAIL` is configured in rtos_cfg.h by setting the definition of `RTOS_CFG_MBOX_DATA`. This allows the application to specify which type of data will be held in the mailbox (i.e. four-byte unsigned, pointer to array of data, etc.). The number of mailboxes available in the application is set by the definition of `RTOS_CFG_NUM_MAILBOX` also defined in rtos_cfg.h. Specific mailbox structures are then referenced by an integer index, where the available integers begin at zero and go to (`RTOS_CFG_NUM_MAILBOX` – 1). Examples of this are found in the API descriptions in the following sections.

3.3.2 Module Public Functions

3.3.2.1 Function *mail_OSmbox_getMail*

Definition:

```
MAIL mail_OSmbox_getMail(U1 mailbox, U4 blockPeriod, U1* errorCode);
```

Purpose:

This function retrieves data from the specified mailbox. Receiving the data also clears the mailbox to zero. If a task is pending to put new data into the mailbox, it is woken up at this time.

Arguments:

`U1 mailbox:` Mailbox number between zero and (`RTOS_CFG_NUM_MAILBOX` – 1).

`U4 blockPeriod:` Timeout period for task blocking measured in scheduler ticks. Set to 0 for non-blocking call.

`U1* errorCode:` Pointer to write error code to.

Return:

`MAIL MBOX_FAILURE` OR
data held in mailbox.

Example:

```
void taskN(void)
{
  U1   u1_t_errorCode;
  MAIL data_t_mail;
  U4   u4_t_getMailTimeout;

  /* Task pend will time out after 5 system ticks. */
  u4_t_getMailTimeout = 5;

  ...

  while(1)
  {
    ...

    /* Store data in mailbox 1 in local variable. Clears mailbox in
       process. */
    u1_t_mailData = mail_OSmbox_getMail(1, u4_t_getMailTimeout,
                                             &u1_t_errorCode);

    ...
  }
}
```

3.3.2.2 Function *mail_OSmbox_checkMail*

Definition:

```
MAIL mail_OSmbox_checkMail(U1 mailbox, U1* errorCode);
```

Purpose:

This function reads the data in the specified mailbox without clearing it. Non-blocking API.

Arguments:

`U1 mailbox:` Mailbox number between zero and (`RTOS_CFG_NUM_MAILBOX` – 1).

`U1* errorCode:` Pointer to write error code to.

Return:

`MAIL MBOX_FAILURE` OR data held in mailbox.

Example:

```
void taskN(void)
{
  U1   u1_t_errorCode;
  MAIL data_t_mail;

  ...

  while(1)
  {
    ...

    /* Store data in mailbox 1 in local variable. Mailbox is not
cleared. */
    u1_t_mailData = mail_OSmbox_checkMail(1, &u1_t_errorCode);

    ...
  }
}
```

3.3.2.3 Function *u1_OSmbox_sendMail*

Definition:

```
U1 u1_OSmbox_sendMail(U1 mailbox, U4 blockPeriod, MAIL data, U1*
                                                         errorCode);
```

Purpose:

Places data into the specified mailbox if the mailbox is empty. If full, blocking is supported.

Arguments:

`U1 mailbox:` Mailbox number between zero and (`RTOS_CFG_NUM_MAILBOX` – 1).

`U4 blockPeriod:` Timeout period for task blocking measured in scheduler ticks. Set to 0 for non-blocking call.

`U1* errorCode:` Pointer to write error code to.

Return:

```
MAIL MBOX_SUCCESS      OR
     MBOX_FAILURE
```

Example:

```
void taskN(void)
{
  U1   u1_t_errorCode;
  MAIL mail_t_data;
  U1   u1_t_sendSts;
  U4   u4_t_sendTimeout;

  ...

  while(1)
  {
    ...

    /* Assume four byte data type. */
    mail_t_data = 0x00FF00FF;

    /* Task pend will time out after 5 system ticks. */
    u4_t_sendTimeout = 5;

    /* Place data in mailbox 1. */
    u1_t_sendSts = u1_OSmbox_sendMail(1, u4_t_sendTimeout, mail_t_data,
                                                          &u1_t_errorCode);

    ...
  }
}
```

3.3.2.4 Function *vd_OSmbox_clearMailbox*

Definition:

```
void vd_OSmbox_clearMailbox(U1 mailbox);
```

Purpose:

Clears data in mailbox. If there is a task pending to send data to this mailbox, then it is woken up at this time.

Arguments:

`U1 mailbox:` Mailbox number between zero and (RTOS_CFG_NUM_MAILBOX – 1).

Return:

```
MAIL MBOX_SUCCESS      OR
     MBOX_FAILURE
```

Example:

```
void taskN(void)
{
  U1   u1_t_errorCode;
  MAIL data_t_mail;

  ...

  while(1)
  {
    ...

    /* Store data in mailbox 1 in local variable. Mailbox is not cleared. */
    u1_t_mailData = mail_OSmbox_checkMail(1, &u1_t_errorCode);

    ...

    /* Reset mailbox data in mailbox 1. */
    vd_OSmbox_clearMailbox(1);
  }
}
```

3.3.3 Module Private Functions

3.3.3.1 Function *vd_OSmbox_init*

Definition:

```
void vd_OSmbox_init(void);
```

Purpose:

This function initializes all mailboxes allocated according to the definition of `RTOS_CFG_NUM_MAILBOX` in rtos_cfg.h. This function is called by the scheduler at startup if the mailbox module is configured.

Arguments:

N/A

Return:

N/A.

Example:

```
void taskN(void)
{
  U1 u1_t_flagSts;

  while(1)
```

```
    {
      ...

      /* Reset flags object and wake all pending tasks. */
      u1_t_flagSts = u1_OSflags_checkFlags(flags1Ptr);

      ...
    }
  }
```

3.3.3.2 Function *vd_OSmbox_blockedTaskTimeout*

Definition:

```
void vd_OSmbox_blockedTaskTimeout(void* mbox);
```

Purpose:

This function exists as a callback for the scheduler. If a task times out while pending on a mailbox, the scheduler calls this function. The mailbox module is then responsible for removing the task from the pending task entry on the mailbox and updating internal data as needed.

Arguments:

N/A

Return:

N/A.

Example:

See sch.c.

3.4 Mutex

The mutex module consists of files mutex.h, mutex_internal_IF.h, and mutex.c. The application should not reference mutex_internal_IF.h, all necessary APIs and definitions for application are in mutex.h.

The mutex module exists for circumstances where mutual exclusion is desired. Its advantage over the semaphore module is that it supports priority inheritance to eliminate cases of priority inversion. The semaphore module still exists for purposes of semaphores that are not mutual exclusion (e.g. counting semaphores). The list manager module is used to maintain internal lists of tasks blocked on each mutex. The mutex module has an internal interface with the scheduler through sch.h and sch_internal_IF.h.

3.4.1 Module Design

The mutex module must be enabled by setting the definition of `RTOS_CFG_OS_MUTEX_ENABLED` to 1 in rtos_cfh.h. Each mutex has the following structure:

```
typedef struct Mutex
{
  U1              lock;             /* Binary lock value (1 is available). */
  BlockedTasks    blockedTaskList;  /* Group for handling blocked tasks. */
  PrioInheritance priority;         /* Group for handling priority
                                       inheritance. */
}
Mutex;
```

The maximum number of mutexes that can exist during runtime is defined by `RTOS_CFG_MAX_NUM_MUTEX` in rtos_cfg.h. The structure `blockedTaskList` contains the information necessary to upkeep a list of tasks blocked on the mutex using the list manager module. The number of tasks that can block on a mutex is set by the definition of `RTOS_CFG_MAX_NUM_BLOCKED_TASKS_MUTEX` also in rtos_cfg.h. This structure is shown below:

```
typedef struct BlockedTasks
{
  /* Memory allocated for storing blocked task data. */
  struct ListNode  blockedTasks[MUTEX_MAX_NUM_BLOCKED];

  /* Pointer to first blocked task in list (highest priority. */
  struct ListNode* blockedListHead;
}
BlockedTasks;
```

The definition of `ListNode` and the APIs used can be found in the list manager header and source files. This list is managed such that the highest priority blocked task on the mutex will be woken up when the mutex becomes available. The second structure in Mutex contains the information needed to support priority inheritance:

```
typedef struct PrioInheritance
{
  U1               taskRealPrio;   /* Real priority of task holding mutex. */
  U1               taskInheritedPrio; /* Inherited priority of task holding
                                         mutex. */
  struct Sch_Task* mutexHolder;    /* Pointer to task holding the mutex. */
}
PrioInheritance;
```

Entry `taskRealPrio` holds the static priority of the task that possesses the mutex at that time. This is used to restore the task's real priority when priority inversion is no longer needed. Similarly, `taskInheritedPrio` stores the inherited priority of the task that possesses the mutex at that time, if the priority is inverted. The pointer `mutexHolder` stores the TCB of the task that possesses the mutex. This pointer is used to request a priority change from the scheduler module through `u1_OSsch_setNewPriority()` through sch_internal_IF.h.

A mutex is created by the application declaring a pointer of type `OSMutex` and passing it to the initialization function. An example of this is in the following section.

3.4.2 Module Public Functions

3.4.2.1 Function *u1_OSmutex_init*

Definition:

```
U1 u1_OSmutex_init(OSMutex** mutex, U1 initValue);
```

Purpose:

This function initializes a mutex and sets `OSMutex` to point to it. The pointer must be declared by the application. This pointer is also used to refer to that mutex in other API calls.

Arguments:

`OSMutex** mutex:` Address of pointer of type `OSMutex` created by the application.

`U1 initValue:` Initial value of mutex (0 or 1).

Return:

`MUTEX_SUCCESS` OR
`MUTEX_NO_OBJECTS_AVAILABLE`

Both `U1`.

Example:

```
static OSMutex *os_tp_mutex1;

...

int main(void)
{
  U1 u1_t_mutex1InitSts;

  ...

  /* Initialize OS. */
  vd_OS_init(APP_SCHED_PERIOD_ONE_MS);

  ...

  /* Initialize mutex to AVAILABLE. */
  u1_OSmutex_init(&os_tp_mutex1, 1);

  ...

  vd_OSsch_start();
}
```

3.4.2.2 Function *u1_OSmutex_lock*

Definition:

```
U1 u1_OSmutex_lock(OSMutex* mutex, U4 blockPeriod);
```

Purpose:

Claims a mutex. If the mutex Is already claimed, blocking is supported. A task blocked on a mutex will time out after `blockPeriod` scheduler ticks. The function call is non-blocking if `blockPeriod` is set to 0.

Priority inheritance is supported. If the task holding the mutex has a lower priority than the blocking task, then the task with the mutex inherits the blocked task's priority. The priority of the task with the mutex will return to its normal value either when it releases the mutex, or when the blocked task times out or is woken up. In other words, the task with the mutex inherits the blocked task's priority so long as the task with the mutex continues to hold the mutex and the blocked task remains in the sleep state. Thus, the scheduler's requirement of no two awake tasks having the same priority is met.

Every call to `u1_OSmutex_lock()` **must** have a subsequent call to `u1_OSmutex_unlock()` by the same task. A task cannot unlock a mutex unless it is holding it.

Arguments:

`OSMutex* mutex:`	Pointer to mutex object.
`U4 blockPeriod:`	Timeout for blocking. If the task isn't woken up before this many scheduler ticks, then the task will time out and wake up.

Return:

`MUTEX_TAKEN` OR
`MUTEX_SUCCESS`

Both `U1`.

Example:

```
void taskN(void)
{
  U1 u1_t_mutex1LockSts;

  ...

  while(1)
  {
    ...

    /* Claim mutex. Timeout on block after 5 scheduler ticks. */
    u1_t_mutex1LockSts = u1_OSmutex_lock(os_tp_mutex1, 5);

    ...

    /* Release mutex. */
    u1_t_mutex1LockSts = u1_OSmutex_unlock(os_tp_mutex1);

    ...

  }
}
```

3.4.2.3 Function *u1_OSmutex_check*

Definition:

```
U1 u1_OSmutex_check(OSMutex* mutex);
```

Purpose:

Checks the state of a mutex without modifying the mutex itself.

Arguments:

`OSMutex* mutex:` Pointer to mutex object.

Return:

`MUTEX_TAKEN` OR
`MUTEX_SUCCESS`

Both `U1.`

`MUTEX_SUCCESS` indicates that the mutex is available.

Example:

```
void taskN(void)
{
  U1 u1_t_mutex1Sts;

  ...

  while(1)
  {
    ...

    /* Check mutex status. */
    u1_t_mutex1Sts = u1_OSmutex_check(os_tp_mutex1);

    ...

  }
}
```

3.4.2.4 Function *u1_OSmutex_unlock*

Definition:

```
U1 u1_OSmutex_unlock(OSMutex* mutex);
```

Purpose:

Release a mutex. The mutex must have been previously claimed by the task calling this function, else the release will be denied.

Arguments:

`OSMutex* mutex:` Pointer to mutex object.

Return:

`MUTEX_SUCCESS` OR
`MUTEX_ALREADY_RELEASED`

Both `U1`.

Example:

See 3.4.2.2: `u1_OSmutex_lock()`

3.4.3 Module Private Functions

3.4.3.1 Function *vd_OSmutex_blockedTimeout*

Definition:

```
void vd_OSmutex_blockedTimeout(struct Mutex* mutex, struct Sch_Task*
                                                              taskTCB);
```

Purpose:

Used by scheduler to notify mutex module when a blocked task has timed out on a mutex object. The mutex module is then responsible for updating the blocked task list on that mutex, while the scheduler is responsible for updating the task sleep/wake state.

Arguments:

`struct Mutex* mutex:` Pointer to mutex object.

`struct Sch_Task* taskTCB:` Pointer to TCB of timed out task.

Return:

N/A

Example:

See sch.c.

3.5 Queue

The queue module is an implementation of thread-safe circular FIFO buffers. It consists of files queue.h, queue_internal_IF.h, and queue.c. The application should not reference queue_internal_IF.h, all necessary APIs and definitions for application are in queue.h.

Queues allow for the exchange of buffers of data between tasks. Unlike the mailbox module, queues support any number of sending and receiving tasks for each queue. In the case that multiple tasks are waiting to send/receive on a full/empty queue, the blocked task with the highest priority will be woken up when the queue becomes available. The memory for the buffers must be allocated at compile-time by the application and then handed off to the RTOS via the initialization API. Each queue is then treated as a circular buffer internally with all necessary overhead handled by the RTOS.

3.5.1 Module Design

The queue module must be enabled by setting the definition of `RTOS_CFG_OS_QUEUE_ENABLED` to 1 in rtos_cfh.h. Each queue has the following structure:

```
typedef struct Queue
{
  Q_MEM*      startPtr;        /* First memory address of FIFO. */
  Q_MEM*      endPtr;          /* Last memory address of FIFO. */
  Q_MEM*      putPtr;          /* Next data sent will be put here. */
  Q_MEM*      getPtr;          /* Next get() call will take data from here. */
  BlockedList blockedTaskList;/* Structure to track blocked tasks. */
}
Queue;
```

The data type `Q_MEM` is defined by setting `RTOS_CFG_BUFFER_DATA` to some data type (e.g. unsigned four-byte data type, void pointer, etc.) in rtos_cfg.h. The maximum number of queues that can be created in runtime must also be defined via `RTOS_CFG_NUM_FIFO` in rtos_cfg.h. Similar to other modules, queues handle task blocking via a BlockedList structure shown below:

```
typedef struct BlockedList
{
  struct ListNode  blockedTasks[RTOS_CFG_MAX_NUM_BLOCKED_TASKS_FIFO];
  struct ListNode* blockedListHead;
}
BlockedList;
```

The maximum number of tasks that can block on each queue must be defined by `RTOS_CFG_MAX_NUM_BLOCKED_TASKS_FIFO` in rtos_cfg.h.

A queue is created by the application creating an array of type `Q_MEM` and then passing the address and size of this array to the initialization API. The queue is then referenced by the application through a unique ID number that can be stored in a variable after initialization.

3.5.2 Module Public Functions

3.5.2.1 Function *u1_OSqueue_init*

Definition:

```
U1 u1_OSqueue_init(Q_MEM* queueStart, U4 queueLength);
```

Purpose:

This function initializes a queue and returns the queue's ID. The buffer itself must be declared by the application. The returned ID number is used to refer to this queue in future API calls.

Arguments:

`Q_MEM* queueStart:` Pointer to start of array allocated by application. Can be any size.

`U4 queueLength:` Size of buffer in terms of `Q_MEM`, not bytes.

Return:

`FIFO_FAILURE` OR
The queue ID number.

Both `U1`.

Example:

```
static U1    buffer1ID;
static Q_MEM buffer1[50];

...

int main(void)
{
  U1 u1_t_mutex1Sts;

  ...

  /* Initialize OS. */
  vd_OS_init(APP_SCHED_PERIOD_ONE_MS);

  ...

  /* Initialize queue and store queue ID number for future use. */
  buffer1ID = u1_OSqueue_init(&testBuffer[ZERO],
                           sizeof(buffer1)/sizeof(Q_MEM));
  ...

  /* Hand control to OS. */
  vd_OSsch_start();
}
```

3.5.2.2 Function *u1_OSqueue_flushFifo*

Definition:

```
U1 u1_OSqueue_flushFifo(U1 queueNum, U1* error);
```

Purpose:

Clears all values in a queue. Also wakes all tasks blocked on the queue.

Arguments:

`U1 u1_queueNum:` Queue ID number set by initialization function `u1_OSqueue_init()`.

`U1* error:` Pointer to write error to. Possible values in queue.h.

Return:

`FIFO_SUCCESS` OR
`FIFO_FAILURE`

Both `U1`.

Example:

```
void taskN(void)
{
  U1 u1_t_buffer1Sts;
  U1 u1_t_buffer1Error;

  ...

  while(1)
  {
    ...

    /* Clear buffer of all entries */
    u1_t_buffer1Sts = u1_OSqueue_flushFifo(buffer1ID,
                                           &u1_t_buffer1Error);

    ...

  }
}
```

3.5.2.3 Function *data_OSqueue_get*

Definition:

```
Q_MEM data_OSqueue_get(U1 queueNum, U4 blockPeriod, U1* error);
```

Purpose:

Gets entry at head of queue. This spot in the queue is cleared in the process. If any tasks are blocked waiting to send data to this queue because it was full, the task with the highest priority is woken up during this API call. If the woken up task is a higher priority than the task currently running, the high priority task will preempt the running task.

Arguments:

`U1 u1_queueNum:`	Queue ID number set by initialization function `u1_OSqueue_init()`.
`U4 blockPeriod:`	After this many scheduler ticks the blocking task will timeout and wake up. Set to 0 for non-blocking function call.
`U1* error:`	Pointer to write error to. Possible values in queue.h.

Return:

`FIFO_FAILURE` OR
Data at head of queue.

Both `Q_MEM`.

Example:

```
void taskN(void)
{
  U1    u1_t_buffer1Error;
  Q_MEM data_t_buffer1Data;

  ...

  while(1)
  {
    ...

    /* Get data at head of queue. Block for 5 ticks if empty. */
    data_t_buffer1Data = data_OSqueue_get(buffer1ID, 5,
                                          &u1_t_buffer1Error);

    ...

  }
}
```

3.5.2.4 Function *u1_OSqueue_getSts*

Definition:

```
U1 u1_OSqueue_getSts(U1 queueNum, U1* error);
```

Purpose:

Get status of queue (full, empty, or ready).

Arguments:

`U1 u1_queueNum:` Queue ID number set by initialization function `u1_OSqueue_init()`.

`U1* error:` Pointer to write error to. Possible values in queue.h.

Return:

```
FIFO_STS_QUEUE_FULL   OR
FIFO_FAILURE          OR
FIFO_STS_QUEUE_EMPTY  OR
FIFO_STS_QUEUE_READY  OR
```

All `U1`.

Example:

```
void taskN(void)
{
  U1    u1_t_buffer1Sts;
  U1    u1_t_buffer1Error;

  ...

  while(1)
  {
    ...

    /* Get status of queue 1. */
    u1_t_buffer1Sts = u1_OSqueue_getSts(buffer1ID, &u1_t_buffer1Error);

    ...

  }
}
```

3.5.2.5 Function *u1_OSqueue_put*

Definition:

```
U1 u1_OSqueue_put(U1 queueNum, U4 blockPeriod, Q_MEM message, U1*
                                                                 error);
```

Purpose:

Put data into buffer. If the buffer is full, then blocking is supported. When a spot in the queue opens, the blocked task with the highest priority will be woken up. If it is higher priority than the running task, the woken up task will preempt the running task.

Arguments:

`U1 u1_queueNum:` Queue ID number set by initialization function `u1_OSqueue_init()`.

`U4 blockPeriod:` After this many scheduler ticks the blocking task will timeout and wake up. Set to 0 for non-blocking function call.

`Q_MEM message:` Data to be put into the buffer.

`U1* error:` Pointer to write error to. Possible values in queue.h.

Return:

```
FIFO_QUEUE_FULL          OR
FIFO_QUEUE_PUT_SUCCESS
```

All `U1`.

Example:

```
void taskN(void)
{
  U1    u1_t_buffer1Sts;
  U1    u1_t_buffer1Error;
  Q_MEM data_t_buffer1Data;

  ...

  /* Assume one byte data type. */
  data_t_buffer1Data = 0xFF;

  ...

  while(1)
  {
    ...

    /* Put one piece of data into buffer 1. Block for 5 ticks if full.
                                                                     */
    u1_t_buffer1Sts = u1_OSqueue_put(buffer1ID, 5, data_t_buffer1Data,
                                                  &u1_t_buffer1Error);

    ...

  }
}
```

3.5.2.6 Function *u4_OSqueue_getNumInFIFO*

Definition:

```
U4 u4_OSqueue_getNumInFIFO(U1 queueNum, U1* error);
```

Purpose:

Returns the number of messages in the specified buffer.

Arguments:

`U1 u1_queueNum:` Queue ID number set by initialization function `u1_OSqueue_init()`.

`U1* error:` Pointer to write error to. Possible values in queue.h.

Return:

`FIFO_FAILURE` OR
The number of entries.

All `U4`.

Example:

```
void taskN(void)
{
  U1 u1_t_buffer1Error;
  U4 u4_t_buffer1NumEntries;

  ...

  while(1)
  {
    ...

    /* Get number of messages in buffer 1. */
    u4_t_buffer1NumEntries = u4_OSqueue_getNumInFIFO(buffer1ID,
                                                     &u1_t_buffer1Error);

    ...

  }
}
```

3.5.3 Module Private Functions

3.5.3.1 Function *vd_OSqueue_blockedTaskTimeout*

Definition:

```
void vd_OSqueue_blockedTaskTimeout(void* queueAddr, struct Sch_Task*
                                                          taskTCB);
```

Purpose:

Used by scheduler to notify queue module when a blocked task has timed out on a queue. The queue module is then responsible for updating the blocked task list on that queue, while the scheduler is responsible for updating the task sleep/wake state.

Arguments:

`Queue* queueAddr:` Pointer to queue structure.

`struct Sch_Task* taskTCB:` Pointer to TCB of timed out task.

Return:

N/A

Example:

See sch.c.

3.6 Semaphore

The semaphore module is an implementation of semaphores that can be used for counting or signaling purposes. It consists of files semaphore.h, semaphore_internal_IF.h, and semaphore.c. The application should not reference semaphore_internal_IF.h, all necessary APIs and definitions for application are in semaphore.h.

Semaphores should not be used for mutual exclusion due to the possibility of priority inversion. The mutex module is more fitting for mutual exclusion purposes. However, the semaphore module also has less overhead and thus executes APIs faster than the mutex module, and so the semaphore module is preferred whenever priority inheritance is not needed. Additionally, the semaphore module supports counting semaphores, whereas mutexes do not support counting.

3.6.1 Module Design

The queue module must be enabled by setting the definition of `RTOS_CFG_OS_SEMAPHORE_ENABLED` to 1 in rtos_cfh.h. Each semaphore has the following structure:

```
typedef struct Semaphore
{
  S1               sema;
  struct ListNode  blockedTasks[SEMA_MAX_NUM_BLOCKED];
  struct ListNode* blockedListHead;
}
Semaphore;
```

Each semaphore can have as many tasks blocked on it as defined by `RTOS_CFG_NUM_BLOCKED_TASKS_SEMA` in rtos_cfg.h and the semaphore module uses `blockedTasks` and `blockedListHead` in conjunction with the list manager module to manage blocked tasks. The variable `sema` stores the semaphore's value, where anything above zero is "available". Aditionally, the maximum number of semaphores must be set by the definition of `RTOS_CFG_NUM_SEMAPHORES` in rtos_cfg.h.

3.6.2 Module Public Functions

3.6.2.1 Function *u1_OSsema_init*

Definition:

```
U1 u1_OSsema_init(OSSemaphore** semaphore, S1 initValue);
```

Purpose:

This function initializes the semaphore pointed to by `semaphore`. The initial value is `initValue`.

Arguments:

`OSSemaphore** semaphore`: Address of semaphore variable created by application.

`S1 initValue`: Initial value of semaphore.

Return:

`SEMA_SEMAPHORE_SUCCESS` OR
`SEMA_NO_SEMA_OBJECTS_AVAILABLE`

Both `U1`.

Example:

```
static OSSemaphore *sema1;

...

int main(void)
{
    U1 u1_t_sema1InitSts;
    /* Init OS with 1ms tick. */
    vd_OS_init(APP_SCHED_PERIOD_ONE_MS);

    ...

    /* Init sema1 with status of "taken". */
    u1_t_sema1InitSts = u1_OSsema_init(&sema1, 0);

    vd_OSsch_start();
}
```

3.6.2.2 Function *u1_OSsema_wait*

Definition:

```
U1 u1_OSsema_wait(OSSemaphore* semaphore, U4 blockPeriod);
```

Purpose:

This function receives a semaphore by decrementing its value. Blocking is supported if the semaphore's value is 0.

Arguments:

`OSSemaphore* semaphore:` Pointer to semaphore variable created by application.

`U4 blockPeriod:` Block time period in ticks. Set to zero for non-blocking call.

Return:

`SEMA_SEMAPHORE_TAKEN` OR
`SEMA_SEMAPHORE_SUCCESS`

Both `U1`.

Example:

```
void taskN(void)
{
  U1 u1_t_sema1RtnSts;

  ...

  while(1)
  {
    ...

    /* Claim semaphore 1. Return immediately if taken. */
    u1_t_sema1RtnSts = u1_OSsema_wait(sema1, 0);

    ...
    }
}
```

3.6.2.3 Function *u1_OSsema_check*

Definition:

```
U1 u1_OSsema_check(OSSemaphore* semaphore);
```

Purpose:

Checks the status of a semaphore without modifying it.

Arguments:

`OSSemaphore* semaphore:` Pointer to semaphore variable created by application.

Return:

`SEMA_SEMAPHORE_TAKEN` OR
`SEMA_SEMAPHORE_SUCCESS`

Both `U1`.

Example:

```
void taskN(void)
{
  U1 u1_t_sema1RtnSts;

  ...

  while(1)
  {
    ...

    /* Check if semaphore 1 is available or not. */
    u1_t_sema1RtnSts = u1_OSsema_wait(sema1);

    ...

  }
}
```

3.6.2.4 Function *vd_OSsema_post*

Definition:

```
void vd_OSsema_post(OSSemaphore* semaphore);
```

Purpose:

Signal a semaphore by incrementing its value.

Arguments:

`OSSemaphore* semaphore:` Pointer to semaphore variable created by application.

Return:

N/A.

Example:

```
void taskN(void)
{
  ...

  while(1)
  {
    ...

    /* SIgnal semaphore 1. */
    vd_OSsema_post(sema1);

    ...

  }
}
```

3.6.3 Module Private Functions

3.6.3.1 Function *vd_OSqueue_blockedTaskTimeout*

Definition:

```
void vd_OSsema_blockedTimeout(struct Semaphore* semaphore, struct Sch_Task*
                                                                  taskTCB);
```

Purpose:

Used by scheduler to notify queue module when a blocked task has timed out on a semaphore. The semaphore module is then responsible for updating the blocked task list on that semaphore, while the scheduler is responsible for updating the task sleep/wake state.

Arguments:

`struct Semaphore* semaphore:` Pointer to semaphore structure.

`struct Sch_Task* taskTCB:` Pointer to TCB of timed out task.

Return:

N/A

Example:

See sch.c.

4 Internal Modules

This section includes some design information for internal modules and how they are used throughout the RTOS.

4.1 List Manager

The list manager module consists of files listMgr_internal.h and listMgr_internal.c. This module is intended for internal RTOS use to provide common APIs for two purposes: management of the data structures in the scheduler module, and for any other module that maintains lists of tasks blocked on its resources. It assumes doubly linked list structures.

4.1.1 Module Design

Each node in the doubly linked lists has the following structure:

```
typedef struct ListNode
{
  struct ListNode* nextNode;     /* Next */
  struct ListNode* previousNode; /* Prev */
  struct Sch_Task* TCB;          /* Data */
}
ListNode;
```

Each node has the typical entries of any doubly linked list node. A pointer to the next node, previous node, and the data. Given the purposes of this module, the data type is a scheduler module TCB structure so that the data pointed to is the necessary task data.

4.1.2 Module Functions

Knowledge of doubly linked lists is assumed, and so the APIs are presented below in a concise format. NULL pointers are properly maintained at head and end nodes:

Function	Return	Argument(s)			Purpose
		Type	Name	Description	
vd_list_addNodeToEnd	void	struct ListNode**	listHead	Address of pointer to head node.	Add new node to end of a linked list.
		struct ListNode*	newNode	Pointer to node to be added.	
vd_list_addTaskByPrio	void	struct ListNode**	listHead	Address of pointer to head node.	Add node to linked list in order of the task priority in its pointed to TCB.
		struct ListNode*	newNode	Pointer to node to be added.	
vd_list_addNodeToFront	void	struct ListNode**	listHead	Address of pointer to head node.	Add node to front of linked list.
		struct ListNode*	newNode	Pointer to node to be added.	
vd_list_removeNode	void	struct ListNode**	listHead	Address of pointer to head node.	Remove a specified node from a linked list.
		struct ListNode*	newNode	Pointer to node to be removed.	
node_list_removeFirstNode	ListNode*	struct ListNode**	listHead	Address of pointer to head node.	Removes head node of a linked list.
node_list_removeNodeByTCB	ListNode*	struct ListNode**	listHead	Address of pointer to head node.	Removes a node specified by the TCB it points to from a linked list.
		struct Sch_Task*	taskTCB	Pointer to TCB stored by node to be removed.	

4.2 CPU/OS Interface

This module consists of files cpu_os_interface.h, cpu_os_interface.c, cpu_defs.h, and taskSwitch.s. This module makes up the Hardware Abstraction Layer (HAL) that exists between the rest of the hardware and the RTOS, so any changes that need to be made for porting are made in these three files.

Its functionality includes the system tick configuration, task stack initialization, context switch, nested critical sections, interrupt enabling/disabling/masking, and register address definitions used by the RTOS. The APIs in this module are not described as they are necessarily changed with any port to a different microcontroller. From the application view, all access to functionality that is not hardware-agnostic is through other RTOS modules, which then internally use this module. Thus any application code is hardware-agnostic with respect to the RTOS and need not change with a port since any interface with the RTOS remains the same.

5 Appendices

Appendix A: Sample application file setting up and running OS.

```
/* OS includes */
#include "sch.h"

/*************************************************************************/
/*  Definitions                                                          */
/*************************************************************************/
#define OS_TICK_MS              (1)
#define OS_TASK_STACK_SIZE      (200)

#define OS_TASK1_PRIO           (1)
#define OS_TASK2_PRIO           (2)
#define OS_TASK3_PRIO           (3)
```

```c
#define OS_TASK1_PERIOD         (1)
#define OS_TASK2_PERIOD         (5)
#define OS_TASK3_PERIOD         (10)

/**********************************************************************/
/*  Private Function Prototypes                                       */
/**********************************************************************/
static void app_task1(void);
static void app_task2(void);
static void app_task3(void);

/**********************************************************************/
/*  Global Variables                                                  */
/**********************************************************************/
static U4 u4_taskStack [OS_TASK_STACK_SIZE];
static U4 u4_taskStack2[OS_TASK_STACK_SIZE];
static U4 u4_taskStack3[OS_TASK_STACK_SIZE];

/**********************************************************************/

/**********************************************************************/
/*  Function Name: main                                               */
/*  Purpose:       Control should be handed to RTOS.                  */
/*                 at end of function.                                */
/*  Arguments:     N/A                                                */
/*  Return:        N/A                                                */
/**********************************************************************/
int main()
{
  /* Initialize OS with tick period. */
  vd_OS_init(OS_TICK_MS);

  /* Init task 1. */ /* Pointer to function definition. */
  u1_OSsch_createTask(&app_task1,
                      /* Pointer to highest memory address in stack. */
                      &u4_taskStack[OS_TASK_STACK_SIZE - 1],
                      /* Size of stack. */
                      sizeof(u4_taskStack),
                      /* Priority of task. */
                      OS_TASK1_PRIO,
                      /* ID number of task for some API calls. Kept same as
                      priority for simplicity. */
                      OS_TASK1_PRIO);

  /* Init task 2. */ /* Pointer to function definition. */
  u1_OSsch_createTask(&app_task2,
                      /* Pointer to highest memory address in stack. */
                      &u4_taskStack2[OS_TASK_STACK_SIZE - 1],
                      /* Size of stack. */
                      sizeof(u4_taskStack2),
                      /* Priority of task. */
                      OS_TASK2_PRIO,
                      /* ID number of task for some API calls. Kept same as
                      priority for simplicity. */
                      OS_TASK2_PRIO);
/* Init task 3. */ /* Pointer to function definition. */
  u1_OSsch_createTask(&app_task3,
                      /* Pointer to highest memory address in stack. */
                      &u4_taskStack3[OS_TASK_STACK_SIZE - 1],
                      /* Size of stack. */
                      sizeof(u4_taskStack3),
                      /* Priority of task. */
                      OS_TASK3_PRIO,
                      /* ID number of task for some API calls. Kept same as
                      priority for simplicity. */
                      OS_TASK3_PRIO);

  /* Hand control to OS, will not return. */
  vd_OSsch_start();
}

/***************************************************************************/
/*  Function Name: app_task1                                               */
/*  Purpose:                                                               */
/*  Arguments:     N/A                                                     */
/*  Return:        N/A                                                     */
/***************************************************************************/
static void app_task1(void)
{
  U4 u4_t_sleepTime;

  u4_t_sleepTime = OS_TASK1_PERIOD;

  while(1)
  {
    //

    vd_OSsch_taskSleep(u4_t_sleepTime);
  }

}

/***************************************************************************/
/*  Function Name: app_task2                                               */
/*  Purpose:                                                               */
/*  Arguments:     N/A                                                     */
/*  Return:        N/A                                                     */
/***************************************************************************/
static void app_task2(void)
{
  U4 u4_t_sleepTime;

  u4_t_sleepTime = OS_TASK2_PERIOD;

  while(1)
  {
    //

    vd_OSsch_taskSleep(u4_t_sleepTime);
  }

}
```

```
/*************************************************************************/
/*  Function Name: app_task3                                            */
/*  Purpose:                                                            */
/*  Arguments:      N/A                                                 */
/*  Return:         N/A                                                 */
/*************************************************************************/
static void app_task3(void)
{
  U4 u4_t_sleepTime;

  u4_t_sleepTime = OS_TASK3_PERIOD;

  while(1)
  {
    //

    vd_OSsch_taskSleep(u4_t_sleepTime);
  }

}
```

Appendix B: Recorded performance metrics.