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 of the scheduler incorporated a layered approach, allowing all RTOS modules to be fully portable to any microcontroller, dependent only on the CPU_OS_Interface internal module. It 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 well.

   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

HuskEOS v2.x Design and User Documentation

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 OS/CPU 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 OS/CPU 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.

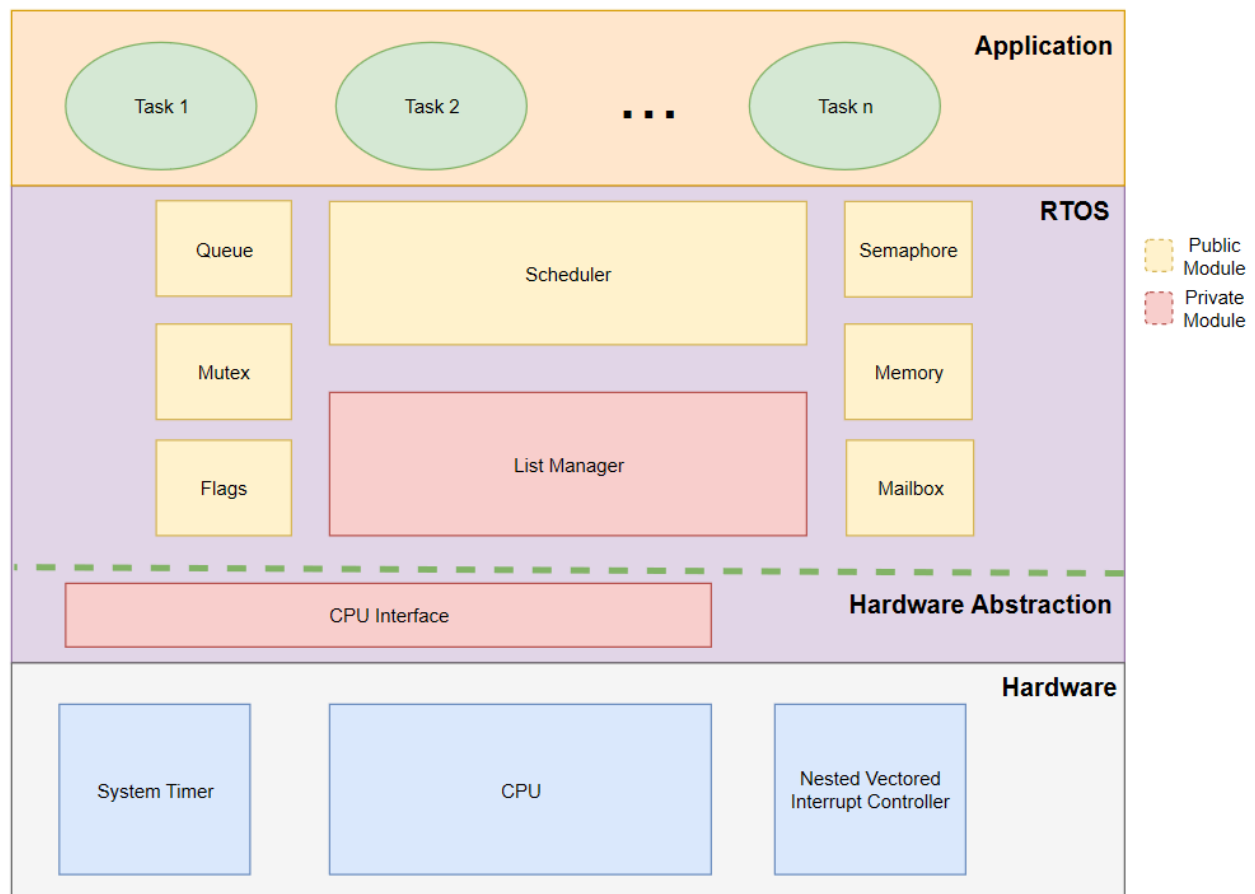This layered view of a potential system is then shown below:

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_OSmodule_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.*

HuskEOS v2.x Design and User Documentation

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 Modules

This section includes some design information and API descriptions for each 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).

HuskEOS v2.x Design and User Documentation

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

```c
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:

```c
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.

HuskEOS v2.x Design and User Documentation

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:
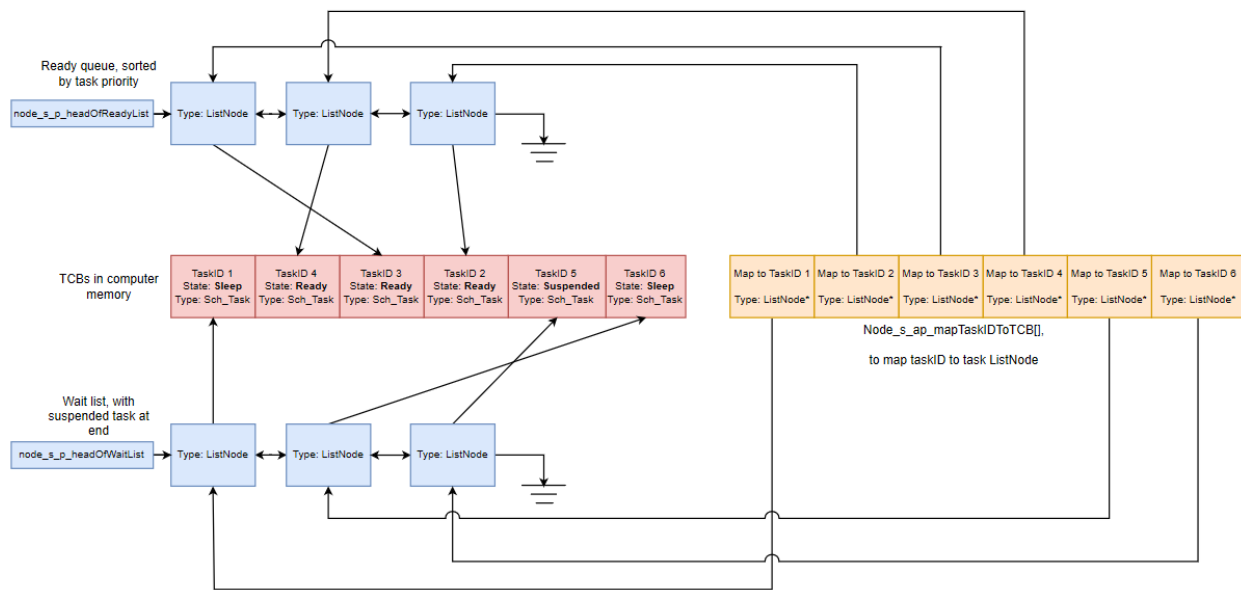

Figure 3.1: Scheduler data structures assuming six tasks exist.

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 OS_CPU_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:

```c
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)
```

HuskEOS v2.x Design and User Documentation

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. */

  ...
```

HuskEOS v2.x Design and User Documentation

```
    /* 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.

HuskEOS v2.x Design and User Documentation

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();

    ...
  }
}
```

HuskEOS v2.x Design and User Documentation

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
```

HuskEOS v2.x Design and User Documentation

```
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();
```

HuskEOS v2.x Design and User Documentation

```
      ...
    }
  }
```

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

HuskEOS v2.x Design and User Documentation

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();

    ...
  }
```

HuskEOS v2.x Design and User Documentation

```
      }
```

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.

HuskEOS v2.x Design and User Documentation

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)
  {
    ...
```

HuskEOS v2.x Design and User Documentation

```
        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.

HuskEOS v2.x Design and User Documentation

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.

3.1.3.1 Function *vd_OSsch_setReasonForWakeup*

Definition:

```c
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:

```c
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);
```

HuskEOS v2.x Design and User Documentation

```
      /* 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);
```

HuskEOS v2.x Design and User Documentation

```
        /* 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();

  ...
}
```

HuskEOS v2.x Design and User Documentation

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.

HuskEOS v2.x Design and User Documentation

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. */
```

HuskEOS v2.x Design and User Documentation

```
          ...

      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);
        ...
```

HuskEOS v2.x Design and User Documentation

```
        }
    }
```

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;
```

HuskEOS v2.x Design and User Documentation

```
      ...

      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:

HuskEOS v2.x Design and User Documentation

Values of flags (U1).

Example:

```c
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:

```c
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 minor extra implementation 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. 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. 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 *vd_OSmbox_init*