

# 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](mailto:gscultho@umich.edu) and/or [djicicala@umich.edu](mailto:djcicala@umich.edu).

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

# 1 Overview

HuskeEOS 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, and also 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:

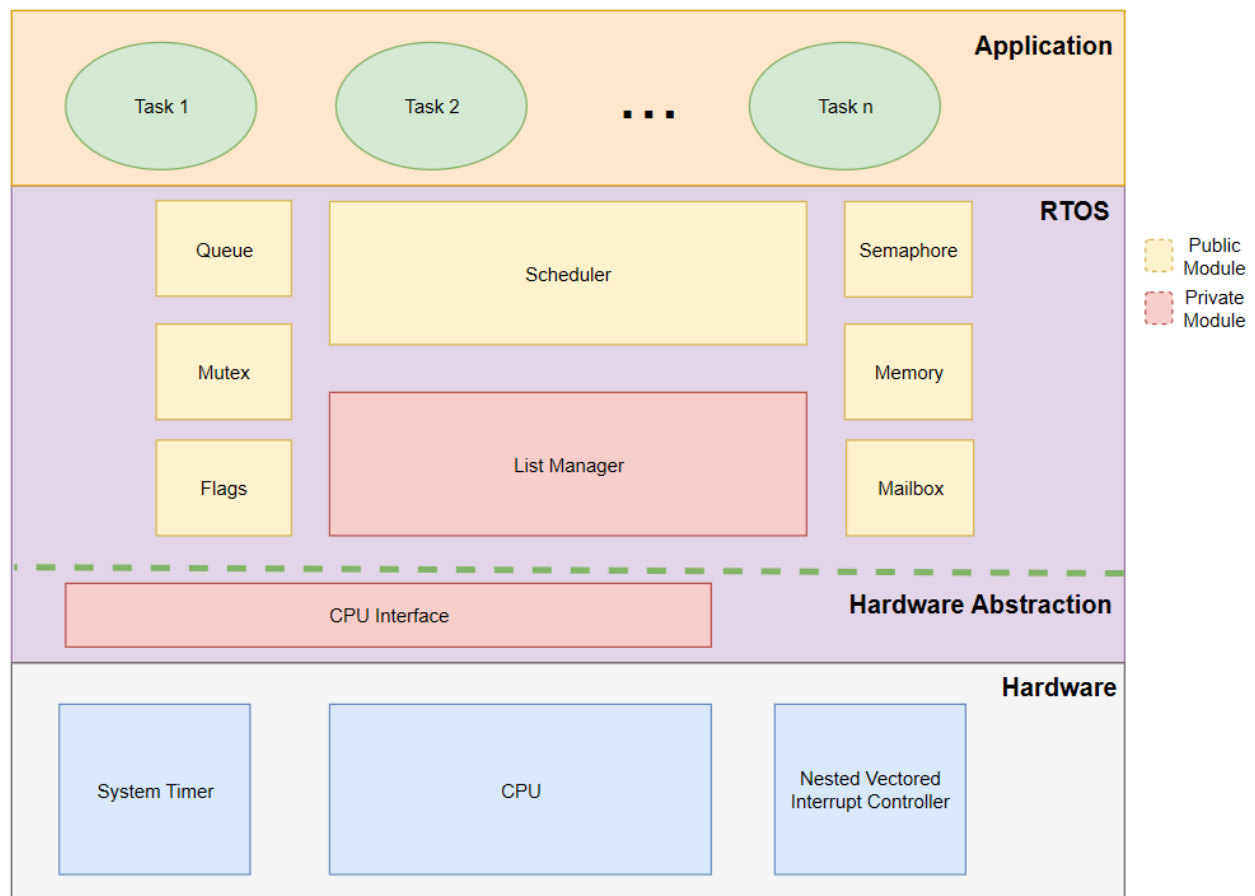


Figure 1.1: Layered architecture approach with HuskeEOS.

It is also seen that other modules can easily be added to the system and even use the RTOS modules as well, such as a communication layer/stack.

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*.

### 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. 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 written for the ARM Cortex-M4 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

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).

#### 3.1.1 Module Design

The scheduler has three simple internal data structures used to schedule tasks, manage their states, and provide task blocking functionality to other modules. Tasks can have three states, being “ready”, “sleeping”, and “suspended”. Sleeping has a timeout, while suspended means that the task is sleeping until explicitly woken up by a function call. The first 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 a pointer 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. */
#ifdef(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 are placed at the front of the linked list while the suspended tasks to 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`. When a task is moved between these two lists, the entire `ListNode` structure (via changing the next/previous pointers) is moved to the list. Thus, each task has one `ListNode` structure in the scheduler.

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` 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 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:

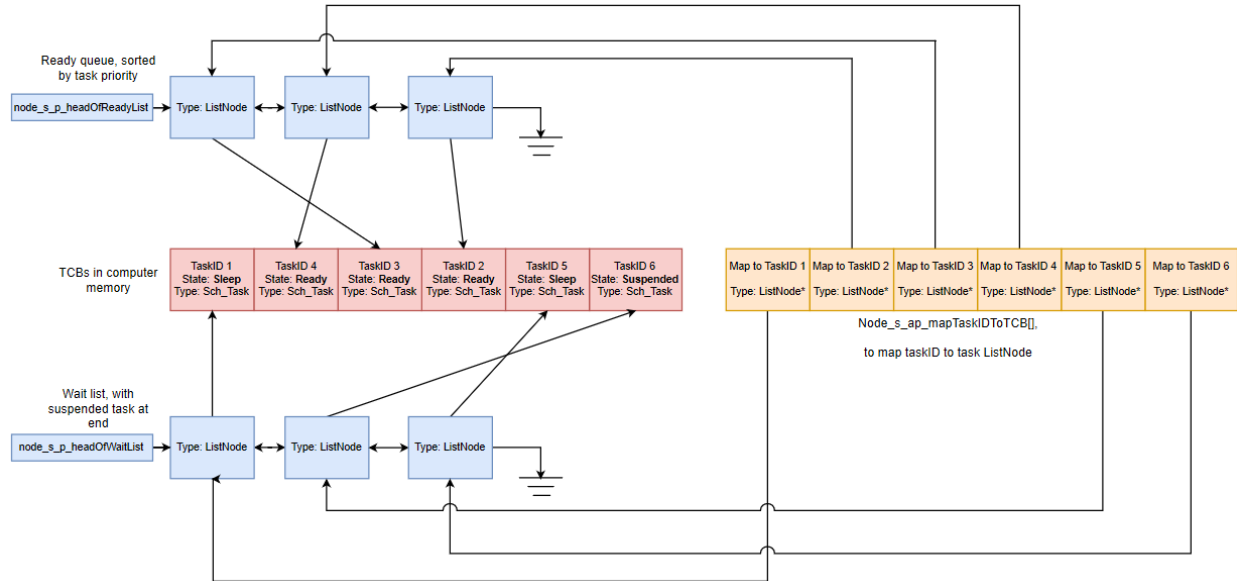


Figure 3.1: Scheduler data structures assuming six tasks exist.

Referencing the diagram above, the tasks exist such that the priority of task with ID 4 > priority of taskID 3 > priority of taskID 2. TaskID is not associated with priority in any way, as seen in the documentation for `u1_OSch_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[]`. This structure exists due to the application referencing the tasks through this taskID number.