Software Driver

OS Abstraction Middleware

Introduction

This Application Note describes the operation of the Renesas OS Abstraction middleware for Renesas microcontrollers. This document does assume that the reader has some knowledge of e^2 studio and CS+.

Target Device

Renesas Microcontrollers

Driver Dependencies

For OS abstraction with an embedded OS, the middleware requires the underlying OS to be within the project.

For OS abstraction without an embedded OS, the middleware requires the OSTM driver to be within the project.

List of Abbreviations and Acronyms

Abbreviation	Full Form
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ISR	Interrupt Service Routine
OS	Operating System
OSTM	Operating System Timer Module
RTOS	Real Time Operating System

Table 1-1 List of Abbreviations and Acronyms

Contents

1. Outline of OS Abstraction 4

2. Description of the Middleware 4
2.1 Structure 4
2.2 Description of each file 4

3. Example of Use 5
3.1 Create Task 5
3.2 Create Mutex 5
3.3 Create Semaphore 5
3.4 Create Event 5
3.5 Create Message Queue 5

4. Module Documentation 6
4.1 Detailed Description 6
4.2 Known Limitations 6
4.3 Known Implementations 6
4.4 Related modules 6
4.5 Macro Definition Documentation 6
4.6 Function Documentation 7

5. Data Structure Documentation 22
5.1 st_os_abstraction_info_t Struct Reference 22

6. OS-Less OS Abstraction 23
6.1 Supported Function API 23
6.2 Connections to external non-API components 24
6.2.1 Compiler Abstraction 24
6.2.2 System Timer 24

7. FreeRTOS OS Abstraction 25
7.1 Supported Function API 25
7.2 Task Priorities 26
7.3 Connections to external non-API components 27
7.3.1 Compiler Abstraction 27
7.3.2 System Timer 27
7.3.3 FreeRTOS 27

Website and Support 28

Revision History 29

1. Outline of OS Abstraction

The OS Abstraction middleware provides the user with a standardized API to operating system features for process and task control.

By using a common, consistent API for OS access, the effort involved with porting application code to different operating systems is greatly simplified. Furthermore, with OS-less OS abstraction, a common approach is provided for non-OS environments as well.

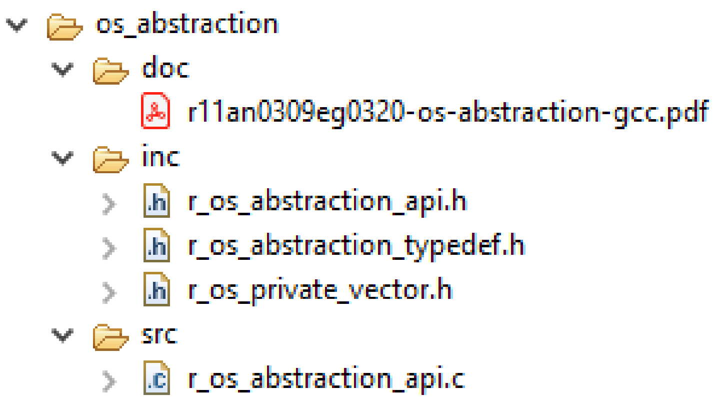
2. Description of the Middleware

The key features to configure:

- Tasks – not used in the OS Less variant of this API
- Mutexes
- Semaphores
- Memory Allocation
- Events
- Message Queues

2.1 Structure

An example of the OS abstraction file structure can be seen in the image below.



```
os_abstraction
    doc
        r11an0309eg0320-os-abstraction-gcc.pdf
    inc
        r_os_abstraction_api.h
        r_os_abstraction_typedef.h
        r_os_private_vector.h
    src
        r_os_abstraction_api.c
```

2.2 Description of each file

Each file's description can be seen in the following table.

Filename	Usage	Description
r_os_abstraction_api.h	To be included in any file which executes the OS Abstraction API	This and r_task_priority.h are the only API header files to include in application code
r_os_private_vector.h	System Configuration only	System Configuration only
r_os_abstraction_typedef.h	Included by r_os_abstraction_api.h	Defines OS abstraction data types
r_task_priority.h	Included by the application	Task priority definitions. Not required if OS-less OS abstraction is used.
r_os_abstraction_api.c	Private	The OS abstraction code implementation.

3. Example of Use

This section describes a simple example of creating a task, mutex, semaphore, event and message queue.

3.1 Create Task

```
os_task_t * p_os_task;

p_os_task = R_OS_TaskCreate(“My Task”, my_task_function, NULL,
   R_OS_ABSTRACTION_SMALL_STACK_SIZE, 6);

if (NULL == p_os_task)
{
   printf(“Task Creation Error”);
}
```

3.2 Create Mutex

```
void *p_mutex = R_OS_MutexCreate();
```

3.3 Create Semaphore

```
uint32_t my_semaphore = 0;
uint32_t count = 10u;
bool_t success;

success = R_OS_SemaphoreCreate((p_semaphore_t) &my_semaphore, count);
if (!success)
{
   printf(“Semaphore Creation Error”);
}
```

3.4 Create Event

```
p_event_t my_event = NULL;
bool_t success;

success = R_OS_EventCreate(&my_event);
if (!success)
{
   printf(“Event Creation Error”);
}
```

3.5 Create Message Queue

```
uint32_t queue_size = 10u;
bool_t success;
p_os_msg_queue_handle_t my_message_queue_handle;

success = R_OS_MessageQueueCreate(&my_message_queue_handle, queue_size);
if (!success)
{
   printf(“Message Queue Creation Error”);
}
```

4. Module Documentation

4.1 Detailed Description

Provides OS abstraction, use these primitives in the code base NOT direct calls to underlying OS primitives.

Provides type defines for OS abstraction.

To make efficient code re-use the identical API shall be used in both OS and OS Less applications. This file aims to abstract the Operating system (OS) awareness when creating an OS Less driver.

4.2 Known Limitations

NONE

4.3 Known Implementations

NONE_YET

4.4 Related modules

See also: DS_BOARD_SUPPORT, RZA1H_RSK_OSTM_DRIVER, RZA1H_RSK_LED

4.5 Macro Definition Documentation

#define SRC_RENESAS_APPLICATION_INC_R_OS_ABSTRACTION_API_H_

#define R_OS_ABSTRACTION_VERSION_MAJOR (1)

#define R_OS_ABSTRACTION_VERSION_MINOR (0)

#define R_OS_ABSTRACTION_UID (81)

#define R_OS_ABSTRACTION_BUILD_NUM (0)

Build Number of API.
Generated during customer release.

#define R_OS_ABSTRACTION_EV_WAIT_INFINITE (0xFFFFFFFFUL)

Maximum timeout used in wait functions inside the OS abstraction module

#define R_OS_ABSTRACTION_INVALID_HANDLE (-1)

Invalid handle used in functions inside the OS abstraction module

#define R_OS_ABSTRACTION_TINY_STACK_SIZE (0)

Stack sizes, these indexes are mapped to actual sizes inside the OS abstraction module

#define R_OS_ABSTRACTION_SMALL_STACK_SIZE (1)

#define R_OS_ABSTRACTION_DEFAULT_STACK_SIZE (2)

#define R_OS_ABSTRACTION_LARGE_STACK_SIZE (3)

#define R_OS_ABSTRACTION_HUGE_STACK_SIZE (4)

#define R_OS_ABSTRACTION_MAX_TASK_NAME_SIZE (24)

#define R_OSFREE_MAX_MUTEXES (32)

Max number of simultaneous mutexes available. Adjust to suit application

#define R_OSFREE_MAX_EVENTS (32)

Max number of simultaneous events available. Adjust to suit application

#define R_OS_ABSTRACTION_OSTM_RESOURCE ("\\\\.\\ostm_reserved")

#define R_OS_MS_TO_SYSTICKS(n) (n)

#define R_OS_SYSTICKS_TO_MS(n) (n)

4.6 Function Documentation

bool_t R_OS_AbstractionLayerInit (void)

Function to configure critical resources for the connected OS or scheduler.

Return values:

true	if there were no errors when initialising the OS Abstraction Layer.
false	if there errors when initialising the OS Abstraction Layer.

bool_t R_OS_AbstractionLayerShutdown (void)

Function to release critical resources for the connected OS or scheduler.

Return values:

true	if there were no errors when closing the OS Abstraction Layer.
false	if there errors when closing the OS Abstraction Layer.

void R_OS_AssertCalled (volatile const char * *p_file*, volatile uint32_t *line*)

Generic error handler, enters forever loop but allows debugger to step out..

Parameters:

in	*file*	file in which the error occurred.
in	*line*	line where the error occurred.

Return values:

NONE.	

void R_OS_EnterCritical (void)

Enter critical area of code - prevent context switches.

OS Abstraction R_OS_EnterCritical Function bool_t R_OS_EventCreate (pp_event_t *pp_event*)

Create an event object for inter-task communication.

Parameters:

in	*pp_event*	Pointer to an associated event.

Returns:

The function returns TRUE if the event object was successfully created. Otherwise, FALSE is returned

void R_OS_EventDelete (pp_event_t *pp_event*)

Delete an event, freeing any associated resources.

Parameters:

in	*pp_event*	Pointer to an associated event.

Returns:

none

e_event_state_t R_OS_EventGet (pp_event_t *pp_event*)

Returns the state on the associated event.

Parameters:

in	*pp_event*	Pointer to an associated event.

Return values:

EV_RESET	Event Reset.
EV_SET	Event Set.
EV_INVALID	Invalid Event.

void R_OS_EventReset (pp_event_t *pp_event*)

Clears the state on the associated event. Setting event to EV_RESET.

Parameters:

in	*pp_event*	Pointer to a associated event.

Returns:

none.

void R_OS_EventSet (pp_event_t *pp_event*)

Sets the state on the associated event outside of an interrupt service routine. Setting event to EV_SET.

Parameters:

in	*pp_event*	Pointer to an associated event.

Returns:

none.

bool_t R_OS_EventSetFromIsr (pp_event_t *pp_event*)

Sets the state on the associated event from inside an interrupt service routine. Setting event to EV_SET

Warning:

Function shall only be called from within an ISR routine

Parameters:

in	*pp_event*	Pointer to an associated event

Returns:

The function returns TRUE if the event object was successfully set. Otherwise, FALSE is returned

bool_t R_OS_EventWait (pp_event_t *pp_event*, systime_t *timeout*)

Blocks operation until one of the following occurs

A timeout occurs.

The associated event has been set.

Parameters:

in	*pp_event*	Pointer to an associated event.
in	*timeout*	Maximum time to wait for associated event to occur.

Returns:

The function returns TRUE if the event object was set, Otherwise, FALSE is returned

void R_OS_Free (void ** *pp_memory_to_free*)

Function to free allocated memory.

Parameters:

in	*p_memory_to_free*	Block of memory to free.

Returns:

None.

uint32_t R_OS_GetTickCount (void)

Gets ticks currently counted for task which calls it.

Warning:

Function can only be called when the scheduler is running

Returns:

The function returns the number of ticks counted.

int32_t R_OS_GetVersion (st_os_abstraction_info_t * *p_info*)

Obtains the version information from this module.

Parameters:

in	*p_info*	Structure containing module version information.

Returns:

The function returns 0

void R_OS_KernelInit (void)

Function to configure critical resources for the connected OS or scheduler, or configure an OS-Less sample.

Return values:

NONE.	

void R_OS_Running (void)

Function used to determine if the connected OS or scheduler has started.

Return values:

TRUE	Scheduler has started
FALSE	Scheduler has not started

void R_OS_KernelStart (void)

Function to enable the connected OS or scheduler, or configure an OS-Less sample.

Return values:

NONE.	

void R_OS_KernelStop (void)

Function to stop the connected OS or scheduler, or configure an OS-Less sample. Provided for completeness, may never be used. When powering down a system safely this function should be called.

Return values:

NONE.	

void* R_OS_Malloc (size_t *size*, e_memory_region_t *region*)

Allocates block of memory the length of "size".

Parameters:

in	*size*	Size of memory to allocate.
in	*region*	Region of memory to allocate from.

Returns:

Allocated memory

bool_t R_OS_MessageQueueClear (p_os_msg_queue_handle_t p_queue_handle)

Clear a message queue, resetting it to an empty state.

Parameters:

in	p_queue_handle	pointer to queue handle.

Returns:

The function returns TRUE if the event object was successfully cleared. Otherwise, FALSE is returned

bool_t R_OS_MessageQueueCreate (p_os_msg_queue_handle_t * *pp_queue_handle*, uint32_t *queue_sz*)

Create a Message Queue of length “queue_sz”.

Parameters:

in	*queue_sz*	Maximum number of elements in queue.
in	*pp_queue_handle*	pointer to queue handle pointer.

Return values:

TRUE	The message queue was successfully created
FALSE	The message queue creation failed.

bool_t R_OS_MessageQueueDelete (p_os_msg_queue_handle_t * *pp_queue_handle*)

Delete a message queue. The message queue pointer argument will be set to NULL.

Parameters:

in	*pp_queue_handle*	pointer to queue handle pointer.

Returns:

The function returns TRUE if the event object was successfully deleted. Otherwise, FALSE is returned

bool_t R_OS_MessageQueueGet (p_os_msg_queue_handle_t *p_queue*, p_os_msg_t * *pp_msg*, uint32_t *timeout*, bool_t *blocking*)

Retrieve a message from a queue. Can only be called outside of an Interrupt Service Routine.

Parameters:

in	*p_queue*	pointer to queue handle.
out	*pp_msg*	pointer to message pointer. Pointer will point to NULL if no message and times out.
in	*timeout*	in system ticks.
in	*blocking*	true = block thread/task until message received. False = not blocking

Returns:

The function returns TRUE if the event object was successfully retrieved from the queue. Otherwise, FALSE is returned

bool_t R_OS_MessageQueuePut (p_os_msg_queue_handle_t *p_queue_handle*, p_os_msg_t *p_message*)

Put a message onto a queue. Can be called from both inside and outside of an Interrupt Service Routine.

Parameters:

in	*p_queue_handle*	pointer to queue handle.
in	*p_message*	pointer to message.

Returns:

The function returns TRUE if the event object was successfully added to the queue. Otherwise, FALSE is returned void R_OS_MutexAcquire (p_mutex_t *p_mutex*)

Acquires possession of a Mutex, will context switch until free, with no timeout.

Parameters:

in	*p_mutex*	Mutex object to acquire.

Returns:

None.

void* R_OS_MutexCreate (void)

Creates a mutex and returns a pointer to it.

Return values:

p_mutex	Pointer to mutex created.
NULL	If mutex creation fails.

void R_OS_MutexDelete (pp_mutex_t *pp_mutex*)

Deletes a Mutex.

Parameters:

in	*pp_mutex*	Address of mutex object to delete, set to NULL when deleted.

Returns:

None.

void R_OS_MutexRelease (p_mutex_t *p_mutex*)

Releases possession of a mutex.

Parameters:

in	*p_mutex*	Mutex object to release.

Returns:

None.

bool_t R_OS_MutexWait (pp_mutex_t *pp_mutex*, uint32_t *time_out*)

Attempts to claim mutex for 'timeout' length, will fail if not possible. If mutex passed is NULL, this function will create new mutex.

Parameters:

in	*pp_mutex*	Address of mutex object to acquire.
in	*time_out*	Length of Time to wait for mutex.

Return values:

TRUE	Mutex is acquired
FALSE	Wait Timed out, mutex not acquired.

bool_t R_OS_SemaphoreCreate (p_semaphore_t *p_semaphore*, uint32_t *count*)

Create a semaphore.

Parameters:

in	*p_semaphore*	Pointer to an associated semaphore.
in	*count*	The maximum count for the semaphore object. This value must be greater than zero

Return values:

TRUE	The semaphore object was successfully created.
FALSE	Semaphore not created.

void R_OS_SemaphoreDelete (p_semaphore_t *p_semaphore*)

Delete a semaphore, freeing any associated resources.

Parameters:

in	*p_semaphore*	Pointer to an associated semaphore.

Returns:

None.

void R_OS_SemaphoreRelease (p_semaphore_t *p_semaphore*)

Release a semaphore, freeing it to be used by another task.

Parameters:

in	*p_semaphore*	Pointer to an associated semaphore.

Returns:

None.

bool_t R_OS_SemaphoreWait (p_semaphore_t *p_semaphore*, systime_t *timeout*)

Blocks operation until one of the following occurs

A timeout occurs.

The associated semaphore has been set.

Parameters:

in	*p_semaphore*	Pointer to an associated semaphore.
in	*timeout*	Maximum time to wait for associated event to occur.

Return values:

TRUE	The semaphore object was successfully set.
FALSE	Semaphore not set.

int_t R_OS_SysLock (void)

Function to lock a critical section.

Warning:

This function must prevent the OS or scheduler from swapping context. This is often implemented by preventing system interrupts form occurring, and so pending any OS timer interruptions. Timing is critical, code protected by this function must be able to complete in the minimum time possible and never block.

Return values:

1	Critical Section entered
0	Object locked
-1	Error, neither action possible

void R_OS_SysReleaseAccess (void)

Function to release system mutex.

The OS Abstraction layer contains a system mutex. This function allows a user to release the mutex from system critical usage.

Returns:

None.

void R_OS_SysUnlock (void)

Function to unlock a critical section.

Warning:

This function releases the OS or scheduler to normal operation. Timing is critical, code proceeding this function must be able to complete in the minimum time possible and never block.

Returns:

None.

void R_OS_SysWaitAccess (void)

Function to acquire system mutex.

The OS Abstraction layer contains a system mutex. This function allows a user to obtain the mutex for system critical usage.

Returns:

None.

os_task_t* R_OS_TaskCreate (const char_t * *p_name*, os_task_code_t *task_code*, void * *p_params*, size_t *stack_size*, int_t *priority*)

Function to create a new task.

Parameters:

in	*p_name*	ASCII character representation for the name of the Task.

Warning:

name string may be subject to length limitations. There is a security risk if the name is not bounded effectively in the implementation.

Parameters:

in	*task_code*	Function pointer to the implementation of the Task.
in	*p_params*	Structure to be used by the Task.
in	*stack_size*	Stack size for allocation to the Task.
in	*priority*	Task priority in system context.

Return values:

os_task_t	The task object.

void R_OS_TaskDelete (os_task_t ** *p_task*)

Function to delete a task.

Warning:

The target OS is responsible for verifying the Task is valid to delete.

Parameters:

in	*p_task*	the task object.

Return values:

None.	

os_task_t* R_OS_TaskGetCurrentHandle (void)

Gets current task.

Warning:

Function shall only be called when the scheduler is running

Parameters:

in	*none*	

Returns:

The function returns the current running task

const char* R_OS_TaskGetCurrentName (void)

Gets text name of current task.

Warning:

Function shall only be called when the scheduler is running

Parameters:

in	*none*	

Returns:

The function returns a pointer to the text name of the current task

int32_t R_OS_TaskGetPriority (uint32_t *task_id*)

Gets current task priority.

Warning:

Function shall only be called when the scheduler is running

Parameters:

in	*task_id*	desired Task

Returns:

The function returns the task priority of the specified uiTaskID
-1 if the uiTaskID can not be found

const char* R_OS_TaskGetState (const char * *p_task*)

Gets status information on selected task in human readable form.

Warning:

Function shall only be called when the scheduler is running

Parameters:

in	*p_task*	task name in human readable form.

Returns:

The function returns a character string that can be displayed on a console.

bool_t R_OS_TaskResume (os_task_t * *p_task*)

Function to cause a task to suspend and pass control back to the OS / scheduler.

Parameters:

in	*task*	the task object.

Return values:

None.	

bool_t R_OS_TaskSetPriority (uint32_t *task_id*, uint32_t *priority*)

Sets current task priority.

Warning:

Function shall only be called when the scheduler is running

Parameters:

in	*task_id*	desired task
in	*Priority*	desired priority

Returns:

true if priority is set
false if priority can not be set

uint32_t R_OS_TasksGetNumber (void)

Function to obtain total number of active tasks defined in the system, only attempted if the operating system is running.

Return values:

Number of tasks

void R_OS_TaskSleep (uint32_t *sleep_ms*)

Function to cause a task to suspend and pass control back to the OS / scheduler for a requested period.

Warning:

The time stated is a minimum, higher priority tasks may prevent this Task form being restored immediately.

Parameters:

in	*sleep_ms*	Time in ms (uint32 => max ~49 Days).

Return values:

None.	

void R_OS_TasksResumeAll (void)

Resume all tasks, only attempted if the operating system is running.

Parameters:

None.	

Return values:

None.	

void R_OS_TasksSuspendAll (void)

Suspend all tasks, only attempted if the operating system is running.

Parameters:

None.	

Return values:

None.	

bool_t R_OS_TaskSuspend (os_task_t * *p_task*)

Function to cause a task to suspend and pass control back to the OS / scheduler.

Parameters:

in	*p_task*	the task object.

Return values:

None.	

void R_OS_TaskUsesFloatingPoint (void)

Function to indicate to the OS that the current task uses floating point numbers.

Return values:

NONE.	

void R_OS_TaskYield (void)

Function to cause a task to suspend and pass control back to the OS / scheduler.

Return values:

None.	

5. Data Structure Documentation

5.1 st_os_abstraction_info_t Struct Reference

```
#include <r_os_abstraction_typedef.h>
```

Data Fields

```
union {
  uint32_t full
  struct {
    uint16_t minor
    uint16_t major
  } sub
} version
uint32_t build
const char * p_szdriver_name
```

Field Documentation

uint32_t build

Build Number Generated during the release

uint32_t full

Major + Minor combined as 1 uint32_t data member

uint16_t major

Version, modified by developer

uint16_t minor

Version, modified by Product Owner

const char* p_szdriver_name

struct { ... } sub

union { ... } version

The documentation for this struct was generated from the following file:

- **r_os_abstraction_typedef.h**

6. OS-Less OS Abstraction

The OS-less OS Abstraction is designed to provide some of the functionality of an OS to a non-OS environment. As it uses the common OS abstraction API, the task of porting between OS based and non-OS applications is simplified.

6.1 Supported Function API

The OS-less OS abstraction supports a reduced subset of the OS abstraction API. Table 6-1 below describes a list the OS abstraction functions and their status. Note that attempts to use unsupported functions will result in an “assert” handled error. These functions can be completed by the developer in an individual application specific way if portability is needed.

Function	Supported	Comments
R_OS_AbstractionLayerInit	✓	Implemented. Starts system timer if not already started.
R_OS_AbstractionLayerShutdown	✓	Implemented. Stops system timer if not already stopped.
R_OS_KernelInit	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_Running	✓	Implemented. Always returns TRUE.
R_OS_KernelStart	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_KernelStop	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_InitMemManager	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TaskCreate	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TaskDelete	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TaskSleep	✓	Wait for specified number of OS timer ticks.
R_OS_TaskYield	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TaskSuspend	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TaskResume	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TasksSuspendAll	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TasksResumeAll	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TasksGetNumber	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TaskUsesFloatingPoint	✗	Returns without doing anything. Does not call assert function
R_OS_TaskGetPriority	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TaskSetPriority	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TaskGetCurrentHandle	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TaskGetCurrentName	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_TaskGetState	✗	Not implemented. Calls assert function R_OS_AssertCalled
R_OS_SysLock	✓	Disables Interrupts
R_OS_SysUnlock	✓	Enables Interrupts
R_OS_SysWaitAccess	✗	Returns without doing anything. Does not call assert function
R_OS_SysReleaseAccess	✗	Returns without doing anything. Does not call assert function
R_OS_GetTickCount	✓	Returns current system tick count.
R_OS_AssertCalled	✓	Places execution into an infinite loop after recording file and line number. Can be used for debug purposes.
R_OS_Malloc	✓	Ignores region parameter and uses system malloc
R_OS_Free	✓	Uses system free

Function	Supported	Comments
R_OS_SemaphoreCreate	✓	
R_OS_SemaphoreDelete	✓	
R_OS_SemaphoreWait	✓	
R_OS_SemaphoreRelease	✓	
R_OS_MutexCreate	✓	The #define R_OSFREE_MAX_MUTEXES defines the number of mutexes available in the system. This can be adjusted to suit the application.
R_OS_MutexDelete	✓	
R_OS_MutexAcquire	✓	
R_OS_MutexRelease	✓	
R_OS_MutexWait	✓	
R_OS_EnterCritical	✓	Disables Interrupts
R_OS_ExitCritical	✓	Enables Interrupts
R_OS_MessageQueueCreate	✓	
R_OS_MessageQueuePut	✓	
R_OS_MessageQueueGet	✓	
R_OS_MessageQueueClear	✓	
R_OS_MessageQueueDelete	✓	
R_OS_EventCreate	✓	The #define R_OSFREE_MAX_EVENTS defines the number of events available in the system. This can be adjusted to suit the application.
R_OS_EventDelete	✓	
R_OS_EventSet	✓	
R_OS_EventReset	✓	
R_OS_EventGet	✓	
R_OS_EventWait	✓	
R_OS_EventSetFromIsr	✓	
R_OS_GetVersion	✓	

Table 6-1 : API functions in OS-less Abstraction

6.2 Connections to external non-API components

6.2.1 Compiler Abstraction

The OS abstraction layer uses the compiler abstraction in order to access simple assembly commands, as defined in the API in "r_compiler_abstraction_api.h".

6.2.2 System Timer

The OS abstraction middleware uses the OSTM timer peripheral to create the system tick functionality. An Interrupt Service Routine (ISR) function, os_abstraction_isr, is called when the timer counter overflows every millisecond, and this increments the system tick counter.

This is achieved by including the ostm driver, using Smart Configurator to set the OSTM peripheral to the correct channel, interval and ISR function.

7. FreeRTOS OS Abstraction

7.1 Supported Function API

The FreeRTOS OS Abstraction is designed to simplify the task of porting application code between Operating Systems.

The FreeRTOS OS abstraction implements the OS abstraction API as a layer above the FreeRTOS instance in the application project. Table 7-1 below describes a list of the OS abstraction functions and their status.

Function	**Supported FreeRTOS: Amazon**	**Comments**
R_OS_AbstractionLayerInit	✓	Calls R_OS_KernelInit
R_OS_AbstractionLayerShutdown	✓	Calls R_OS_KernelStop
R_OS_KernelInit	✓	Calls R_OS_InitMemManager, creates main_task then calls R_OS_KernelStart
R_OS_Running	✓	Implemented. Implemented state true(yes), false(no)
R_OS_KernelStart	✓	
R_OS_KernelStop	✓	
R_OS_InitMemManager	✓	Initialise heap in freeRTOS
R_OS_TaskCreate	✓	
R_OS_TaskDelete	✓	
R_OS_TaskSleep	✓	
R_OS_TaskYield	✓	
R_OS_TaskSuspend	✓	
R_OS_TaskResume	✓	
R_OS_TasksSuspendAll	✓	
R_OS_TasksResumeAll	✓	
R_OS_TasksGetNumber	✓	
R_OS_TaskUsesFloatingPoint	✓	
R_OS_TaskGetPriority	✓	
R_OS_TaskSetPriority	✓	
R_OS_TaskGetCurrentHandle	✓	
R_OS_TaskGetCurrentName	✓	
R_OS_TaskGetState	✓	
R_OS_SysLock	✓	
R_OS_SysUnlock	✓	
R_OS_SysWaitAccess	✓	
R_OS_SysReleaseAccess	✓	
R_OS_GetTickCount	✓	Returns current system tick count.
R_OS_AssertCalled	✓	Places execution into an infinite loop after recording file and line number data to console. Can be used for debug purposes.

Function	**Supported FreeRTOS or Amazon**	**Comments**
R_OS_Malloc	✓	FreeRTOS variant Supports memory region selection allowing application to select preferred memory region. Amazon Variant Does not support memory region selection.
R_OS_Free	✓	
R_OS_SemaphoreCreate	✓	
R_OS_SemaphoreDelete	✓	
R_OS_SemaphoreWait	✓	
R_OS_SemaphoreRelease	✓	
R_OS_MutexCreate	✓	
R_OS_MutexDelete	✓	
R_OS_MutexAcquire	✓	
R_OS_MutexRelease	✓	
R_OS_MutexWait	✓	
R_OS_EnterCritical	✓	
R_OS_ExitCritical	✓	
R_OS_MessageQueueCreate	✓	
R_OS_MessageQueuePut	✓	
R_OS_MessageQueueGet	✓	
R_OS_MessageQueueClear	✓	
R_OS_MessageQueueDelete	✓	
R_OS_EventCreate	✓	
R_OS_EventDelete	✓	
R_OS_EventSet	✓	
R_OS_EventReset	✓	
R_OS_EventGet	✓	
R_OS_EventWait	✓	
R_OS_EventSetFromIsr	✓	
R_OS_GetVersion	✓	

Table 7-1 : API functions in FreeRTOS OS Abstraction

7.2 Task Priorities

The FreeRTOS OS abstraction has a header file “r_task_priority.h” which is used to define the priorities of system tasks, such as the main task, console, idle task etc.

7.3 Connections to external non-API components

7.3.1 Compiler Abstraction

The OS abstraction layer uses the compiler abstraction in order to access simple assembly commands, as defined in the API in "r_compiler_abstraction_api.h".

7.3.2 System Timer

The OS abstraction middleware uses the OSTM timer peripheral to create the system tick functionality. An Interrupt Service Routine (ISR) function, os_abstraction_isr, is called when the timer counter overflows every millisecond.

This is achieved by including the ostm driver, using Smart Configurator to set the OSTM peripheral to the correct channel, interval and ISR function.

7.3.3 FreeRTOS

The OS abstraction layer uses freeRTOS to implement the functionality. As such it includes the following headers, which should be made available in the project

```
#include "FreeRTOS.h"
#include "FreeRTOSconfig.h"
#include "semphr.h"
#include "queue.h"
#include "task.h"
```

7.3.4 Configuring Memory Regions

Memory can be defined allowing support of multiple non adjacent (non-contiguous) memory regions.

FreeRTOS OS abstraction layer includes an enhanced version of the default heap5.c memory module to support the selection of which region is used in the R_OS_Malloc function. When using the FreeRTOS module, the **e_memory_region_t** parameter is used to specify which region is preferred for the allocation.

Certified Amazon FreeRTOS OS supports the specification of memory regions, but does not support the selection of which region R_OS_Malloc() uses. Amazon certification prohibits the modification of core software. When using the certified Amazon module, the **e_memory_region_t** parameter is ignored.

To configure the memory regions (used in both variants) create the **e_memory_region_t** table in the following file: **generate/system/inc/r_typedefs.h** and define the regions in any **.c** file **(ie main.c).**

Example **r_typedefs.h**

```
#define REPLACE_MEMORY_REGION_ENUM (1) /* Override the default implementation */
typedef enum
{
    R_MEMORY_REGION_DEFAULT = 0,
    R_MEMORY_REGION_NEW,
} e_memory_region_t;
```

Example **main.c**

```
#include "FreeRTOS.h"
#include "r_compiler_abstraction_api.h"
#include "r_os_abstraction_api.h"

HeapRegion_t xHeapRegions[] =
{
    {( uint8_t * ) 0x80080000UL, 0x00060000UL}, /* R_MEMORY_REGION_DEFAULT */
    {( uint8_t * ) 0x800E0000UL, 0x00020000UL}, /* R_MEMORY_REGION_NEW */
    {( uint8_t * ) 0x00000000UL, 0x00000000UL}, /* Terminates the array */
};
```

Website and Support

Renesas Electronics website
https://www.renesas.com/

Inquiries
https://www.renesas.com/contact/

Revision History

		Description	
Rev.	**Date**	**Page**	**Summary**
3.10	21/03/2019	All	Created document to align with OS Abstraction layer V.3.10
3.20	25/07/2019	All	Filename change