eTaskSync User Guide

Version 2.40

For use with eTaskSync versions 3.03 and above

Date: 21-Apr-2016 17:59

Table of Contents

System Overview 4
Introduction 4
Feature Check 4
Packages and Documents 5
Packages 5
Documents 5
Change History 6
eTaskSync Overview 7
Operation 7
Example 1 7
Example 2 8
System Context 9
Tasks 10
Task Overview 10
Task Usage 11
Time Slicing 12
Task Yield 14
Events 15
Events Overview 15
Asynchronous Events 15
Mutexes 17
Mutex Overview 17
Priority Inheritance 18
Usage 20
As a main scheduler 20
In a non RTOS or super loop system 20
In an RTOS system 20
eTS Tick 21
Real Time Characteristics 21
Source File List 22
API File 22
Configuration File 22
Source Code Files 22
Version File 23
Quality Metrics 23
Configuration Options 24
Application Programming Interface 25
System Functions 25
sync_init 26
sync_run 27
sync_run - forever 28
Task Functions 29

sync_task_create 30
sync_task_delete 32
sync_task_get_id 33
sync_task_reschedule 34
sync_task_sleep 35
sync_task_yield 36
Event Functions 37
sync_event_init 38
sync_event_delete 39
sync_event_flags_clear 40
sync_event_flags_get 41
sync_event_flags_set 42
sync_event_flags_set_async 43
Mutex Functions 44
sync_mutex_init 45
sync_mutex_delete 46
sync_mutex_get 47
sync_mutex_put 48
Error Codes 49
Debug Module 50
sync_debug_pf_start 51
sync_debug_pf_stop 52
sync_debug_task_info 53
Integration 54
Architecture-specific Functions 54
sync_context_init 55
sync_context_switch 56
Platform-specific Functions 57
sync_tick_init 57
sync_get_tick_count 58
sync_int_disable 59
sync_int_restore 60
PSP Porting 61

1 System Overview

1.1 Introduction

This guide is for those who want to use eTaskSync as a scheduler to synchronize and coordinate operations in an embedded system. eTaskSync (eTS) is a simple but powerful cooperative scheduler. It is designed specifically to meet the requirements of HCC's embedded middleware solutions, but is suitable for use in a much broader context.

eTaskSync uses a small subset of the typical functions of a standard kernel: tasks, events, and mutexes. It can be used for simple cooperative task scheduling in an embedded system and is particularly suited for embedded systems that require a high level of reliability and availability.

eTaskSync can be executed externally and it is possible to define the maximum number of ticks it runs for. This makes it easy to plan the execution times of HCC middleware with non-OS or run-till completion schedulers. The benefit of this approach for the system designer is that middleware stacks do not block the system.

eTaskSync can be used in products which have no operating system. It is especially useful when using middleware such as USB, File System, and TCP/IP. This makes it easy to integrate middleware into any proprietary software environment, such as a super loop.For a detailed description of eTS, see eTaskSync Overview.

1.2 Feature Check

The main features of eTaskSync are as follows:

- It conforms to the HCC Advanced Embedded Framework.
- It is fully compliant with MISRA-C:2004.
- Prioritized tasks.
- Events – these are used as a signalling mechanism, both between tasks, and from asynchronous sources such as Interrupt Service Routines (ISRs) to tasks.
- Mutexes – these guarantee that, while one task is using a particular resource, no other task can pre-empt it and use the same resource.
- Prioritized scheduling.
- Priority inheritance – this avoids situations where a lower priority task blocks the progress of a higher priority task.
- Time slicing – tasks can be allocated a number of ticks.
- It runs on all 8, 16, and 32 bit micro-controllers.

eTaskSync has these supporting features:

- A small footprint: <2KB code, 100 bytes RAM.
- 100% statement, branch and MC/DC coverage test code (**test_sync.c**).

- The test suite provides reference usage code.
- A debug module to help with task stack analysis and task performance analysis.
- Dynamic and static code analysis reports.
- Ports to a wide range of microcontrollers are available.

1.3 Packages and Documents

Packages

The table below lists the packages that need to be used with this module, and also optional modules which may interact with it, depending on your particular system's design:

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
sync_base	The base eTaskSync package.
sync_test	Test module package for eTaskSync. This includes test code to execute complete statement, branch and MC/DC coverage of eTaskSync. This code can also be used as a usage reference for eTaskSync.
sync_psp_xxx_yyy	Target-specific packages for eTaskSync, where xxx is the architecture (for example, ARM7) and yyy is the toolchain (for example, IAR). Generally HCC provides these target-specific packages, though they can also be custom-developed, as described in Integration.

Documents

For an overview of HCC RTOS software, see Product Information on the main HCC website.

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC eTaskSync User Guide

This is this document.

1.4 Change History

This section includes recent changes to this product. For a complete list of all changes, refer to the file **src /history/sync/sync.txt** in the distribution package.

Version	Changes
3.03	Fixed the following problems: • Previously event flags were lost if a task was waiting for multiple event flags, got any, moved to the ready queue and any of the other event flags it was waiting for got set before scheduling the task. • Invalidated owner task for deleted mutex. • An incorrect **sync_run()** execution time could occur if there was no active task. • In case of priority inheritance, the owner task of the mutex could act incorrectly if it was waiting for another mutex.
3.02	Code reorganized and cleaned up. This fixed the following problems: • A priority inversion issue: a task could remain at a higher priority. • A mutex can now only be released by its owner.

2 eTaskSync Overview

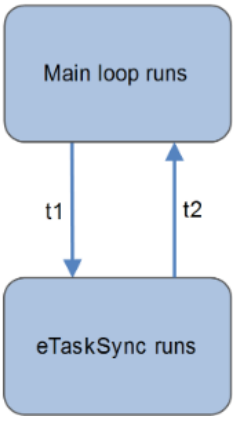
2.1 Operation

eTaskSync is a cooperative scheduler with task, event and mutex support. It executes tasks for a specified number of ticks, then returns control to the host system. Context switching between tasks is only performed when the running task calls an eTS function. This means tasks must be designed to call eTS functions often enough to guarantee the timing requirements of the system.

The Tasks section gives a detailed description of how tasks are managed.

Example 1

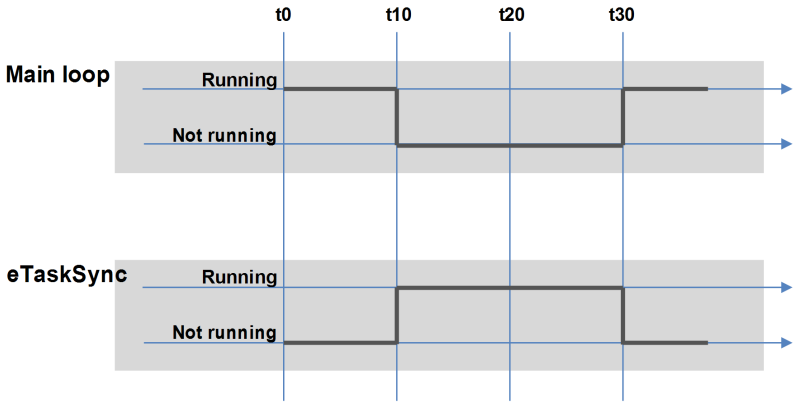
This diagram shows the statechart for a system of which eTaskSync is a subsystem. The system (main loop) calls eTaskSync; this gives control to eTaskSync for the specified number of ticks.



The tasks of the eTaskSync subsystem call eTaskSync functions periodically. As soon as an eTaskSync function is called, it can decide whether to return control to the main loop, because the allocated time has expired, or to run another task.

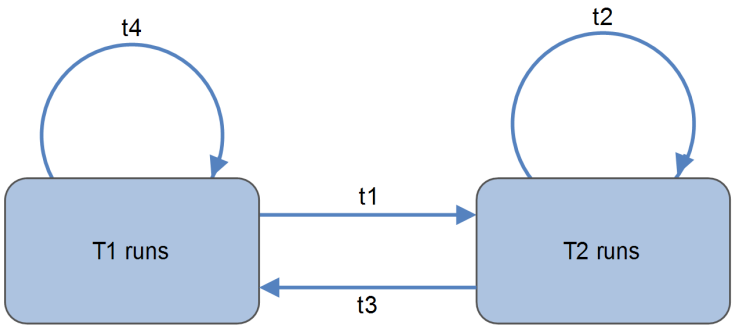
Tick	Transition	Action	Notes
t10	t1	Main loop calls **sync_run(20)**	eTaskSync is called with period of 20 ticks.
t30	t2	One of the following: • eTaskSync time is expired. • Each task is yielded or is in the Wait queue.	eTaskSync returns to the main loop.

The following diagram shows a time sequence diagram for the system.



Example 2

This diagram shows the statechart of an example eTaskSync system, containing two tasks, T1 and T2:



Note the following:

- Only one task can be in the Run state.
- When a task is in the Run state, it only returns control to eTaskSync when it calls an eTaskSync function.
- As soon as an eTS function is called, eTS can decide whether to return control to the main loop because the allocated time has expired, or to run another task.

The following table summarizes the changes to the system. Note that T1 has higher priority.

Transition	Task	Action
t1	T1	T1 goes into the Wait or Delay queue. T2 runs.
t2	T2	eTaskSync reschedules. T2 is ready to run so continues. T1 is in the Wait or Delay queue.
t3	T2	T1 moves to the Ready queue. T2 goes to the Wait/Delay queue or to the Ready queue. T1 runs.
t4	T1	eTaskSync reschedules. T1 is running and continues. T2 is in the Wait, Delay or Ready queue.

2.2 System Context

When you use eTaskSync, always consider the context in which functions are used. This document defines the following three contexts:

- **eTS Context** – this is in a task running in eTS. All eTS functions can be called from this context except eTS system functions.
- **Interrupt Context** – this is when eTS has been interrupted. This could be either in an Interrupt Service Routine or in a high priority in a hosting system that has pre-empted eTS. The only eTS function that can be called from this context is **sync_event_flags_set_async()**.
- **Non-Interrupt Context** – this is when eTS is not running, for instance when **sync_run()** has returned to a main loop. All system functions, resource initialize and delete functions, and **sync_event_flags_set_async()** can be called from this context.

The table below summarizes the functions that can be used in each defined context:

Context	Available Functions
eTS	**sync_task_create()**, **sync_task_delete()**, **sync_task_get_id()**, **sync_task_sleep()**, **sync_task_reschedule()**, **sync_task_yield()**, **sync_event_init()**, **sync_event_delete()**, **sync_event_flags_get()**, **sync_event_flags_set()**, **sync_event_flags_clear**, **sync_event_flags_set_async()**, **sync_mutex_init()**, **sync_mutex_delete()**, **sync_mutex_get()**, **sync_mutex_put**
Interrupt	**sync_event_flags_set_async()**
Non-Interrupt	**sync_init()**, **sync_run()**, **sync_task_create()**, **sync_task_delete()**, **sync_event_init()**, **sync_event_delete()**, **sync_event_flags_set_async()**, **sync_mutex_init()**, **sync_mutex_delete()**

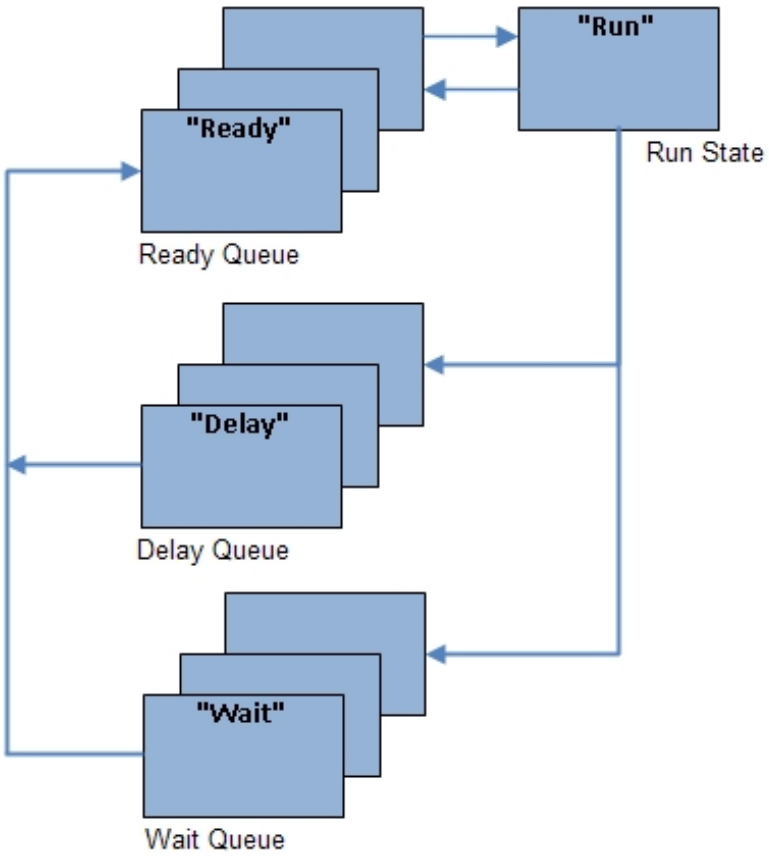
2.3 Tasks

Task Overview

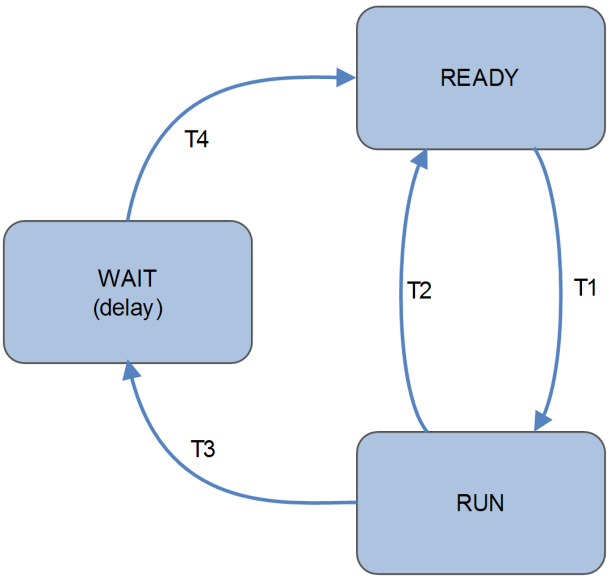
The main purpose of eTaskSync is to schedule a set of tasks based on their priority and state. Each task added to eTS operates entirely within its own context. All interaction with other tasks is performed by using the mutex and event functions. Data and messages are passed between tasks by using global message queues, with access controlled by mutexes and signalling performed by events.

The diagram below shows how tasks are moved between queues in eTS. Note that only one task can be running at any time; the running task must be moved to another queue before another task can be moved to the Run state.

When eTS runs it takes the highest priority task from the Ready queue, moves it to the Run state, and executes it. The running task executes until an eTS function is called by that task, at which point eTS can either move that task to one of the queues and select another task to run from the Ready queue, or return control to the previously running task.



The basic state machine for each task is shown below:



Transition	Description
T1	The task is moved from the Ready queue to the Run state by eTS because it is the highest priority task in the Ready queue.
T2	The task is moved back to the Ready queue by eTS because it has relinquished control, or because its time slice has expired.
T3	The task is moved to the Wait or Delay queue because it is waiting for an event, mutex or timeout.
T4	The task is moved to the Ready queue because the wait or delay condition has been completed, for example an event arrived or a timeout expired.

Task Usage

Tasks are sections of code that run in their own context. The code runs independently of any other code on the system, except through defined communication mechanisms: mutexes and events.

In eTS tasks run until they make a call to any eTS function. When an eTS function is called, the scheduler decides whether control should be returned to that task, or whether other tasks should be run first.

Each task has a priority level. The highest priority task runs first and runs until it relinquishes control, allowing another task to run. Tasks of equal priority are run on a "round robin" basis. A task passes control to the next equal priority task, either when it has gone to wait state, or when its time slice has expired (see Time Slicing).

To ensure all tasks get time to run on the system, and also to ensure that higher priority tasks can pre-empt lower priority tasks, design each task such that:

- During any idle period it calls an eTS function.

- (Particularly for lower priority tasks) eTS functions are called during any code which takes a long time to execute.

The following are key properties:

- Calling **sync_run()** starts the scheduler, which then runs tasks based on their priority and their state.
- Tasks are created by using **sync_task_create()**. As soon as this call returns, the task is moved to the Ready queue.
- An existing task may be deleted at any time using **sync_task_delete()**.
- Any call to an eTS function by a task causes the scheduler to run.
- When the scheduler runs, eTS first checks to see if **sync_run()** time has expired, to determine whether control needs to be returned to the host system. It then handles time slicing and asynchronous events (for example, from ISR), and processes the Ready queue.
- A task can call **sync_task_reschedule()** if it wants to allow higher priority tasks to run. This can be useful for a lower priority task doing something that takes a long time. Intelligent use of this function allows the latency of higher priority tasks to be controlled.
- A task can call **sync_task_yield()** if it wants to allow a lower priority task to run. When this is called, the running task is moved to the end of the Ready queue so that all ready lower priority tasks execute before the task is restored to its previous priority level. If the **sync_run()** exits and is called again, all yielded tasks are restored to the Ready queue at their normal priority, if the lower priority tasks were executed.
- You can determine the unique ID of a task by calling **sync_task_get_id()** within it.
- If a task needs to wait for a period of time before its next action, it can call **sync_task_sleep()** with a specified time period.

Time Slicing

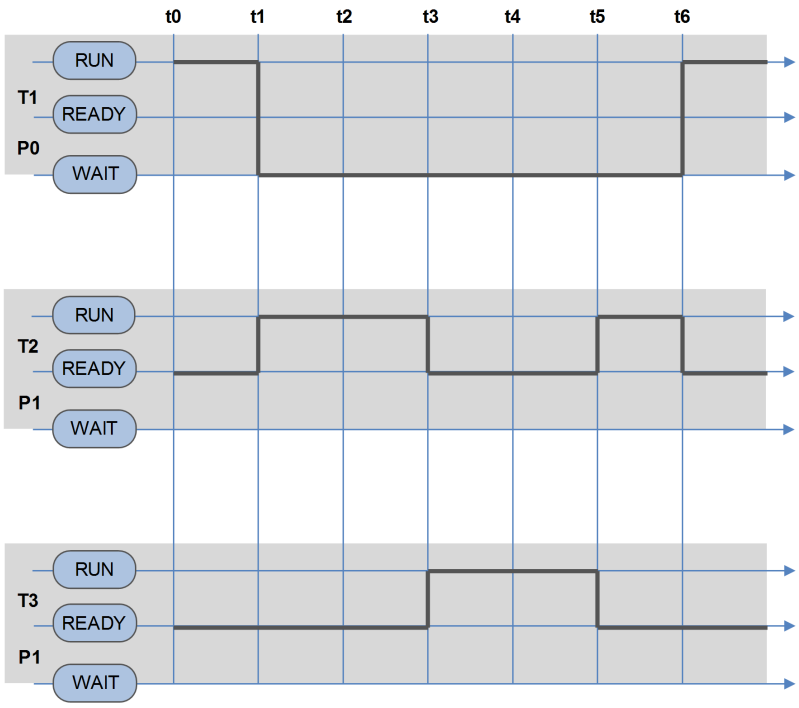
Tasks created in eTaskSync have a priority which increases from Pmin to P0, the highest priority.

If time slicing is disabled, tasks with the same priority are served on a round robin basis. If the running task relinquishes the processor (by calling **sync_yield()**), or waits for a resource, and there is no higher priority task in the ready queue, the next task with the same priority as the running task can run.

If time slicing is enabled, the task descriptor contains a value, time_slice, which holds the number of ticks for the task. Tasks with the same priority run after each other. When the allocated time slice expires, the running task is moved to the Ready queue and the next ready task with the same priority runs.

The diagram below illustrates this mode of operation. (Here, T1 has priority P0. T2 and T3 have priority P1.)

Tick	Task	Action	Notes
t0		initial state	T1 is running. T2 and T3 are in the Ready state. T2 has locked mutex M1. T2 and T3 have their time slice set to 2 ticks.
t1	T1	get_mutex (M1)	T1 waits for mutex M1 and so is moved to the Wait queue. T2 runs.
t2	T2	reschedule()	No change.
t3	T2	reschedule()	Time slice for T2 has expired so it is moved to the Ready queue. T3 runs.
t4	T3	reschedule()	No change.
t5	T3	reschedule()	Time slice for T3 has expired so it is moved to the Ready queue. T2 runs.
t6	T2	put_mutex (M1)	T2 unlocks mutex M1. T1 is moved to the Ready queue. T1 runs as it is the highest priority ready task.

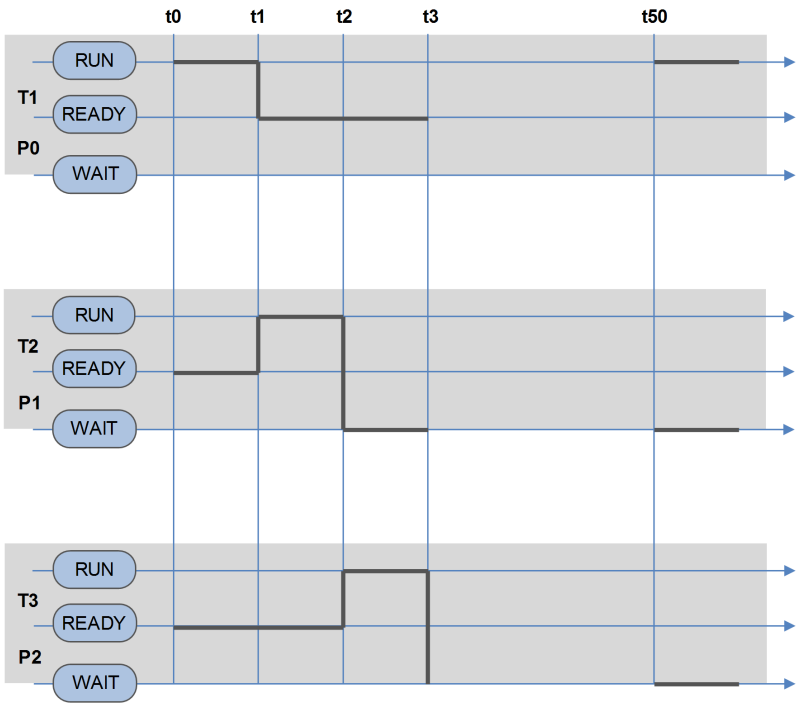


Task Yield

This allows all other non-yielded tasks to run.

The diagram below shows how this function affects the running task. (Here, T1 has priority P0, T2 has P1, and T3 has P2.)

Tick	Task	Action	Notes
t0	T1	Initial State	T1 is running. T2 and T3 are in the Ready queue.
t1	T1	yield()	T1 priority=P(min-1), T2 runs.
t2	T2	get_event (E1)	Event E1 has not been set. T3 runs because T1 has yielded.
t3	T3	get_event (E2)	All tasks are yielded or in the wait queue. eTS returns to the main loop.
t50	Main loop	sync_run()	eTS is called. T1 returns to its normal priority and runs. T2 and T3 are waiting for events.



2.4 Events

Events Overview

Events are used as a signalling mechanism, both between tasks, and from asynchronous sources such as Interrupt Service Routines (ISRs) to tasks.

When a task calls **sync_event_flags_get()**, if that event is not set the task is moved to the Wait queue until that event is satisfied, or until the timeout on that event expires.

Events are organised into groups. Each defined group has 32 flags which you can use as required. The normal method is:

1. Assign a bit define for each event that is to be associated with a particular event group.
2. Use those bits in the **sync_event_flags_get()** call to specify which events should be signalled by the caller.

Likewise, the entity that wants to signal the occurrence of a particular event or set of events within an event group will use the same bit definitions in a **sync_event_flags_set()** call.

For a particular event group, different tasks can wait for different flags, or for the same flags. Each task waiting for a particular flag in an event group receives the event when it is set.

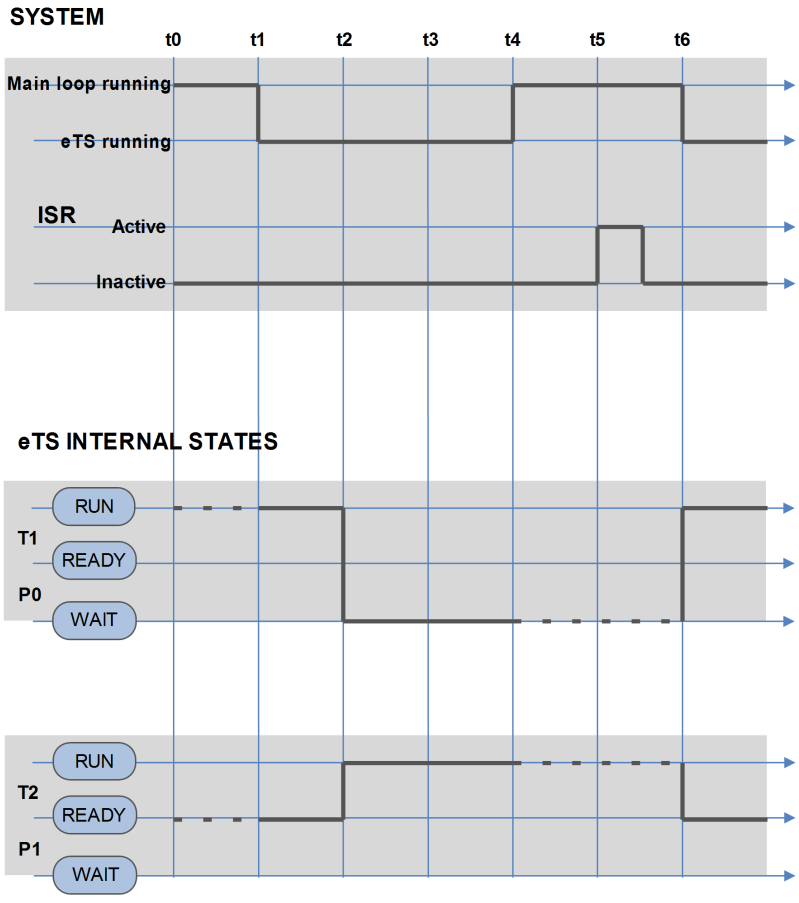
Asynchronous Events

Any Operating System must be able to handle external events. Typically this involves an ISR handling a particular peripheral function; the ISR runs in a special context that does not affect the main system. The ISR needs a method of signalling to the Operating System that a particular event has occurred. This also applies to signalling from any code running outside the eTS context.

In eTaskSync the **sync_event_set_async()** call is provided for any function that is executed in an external context (for example, an ISR or super loop), to signal an event to an eTS task. Because these events are entirely asynchronous to the operation of eTS, they may happen either when eTS is running or when the host system has control. In either case the event generated from an external context cannot be processed until an eTS function is called.

The following diagram shows the scenario when the ISR sets an event while eTS is running. The event generated by the ISR cannot be processed until the currently running task calls an eTS function.

Tick	Task	Action	Notes
t1	main loop	**sync_run(2)**	Main loop calls eTaskSync, parameter is 2 ticks, T1 starts to run.
t2	T1	**get_event(E1)**	T1 waits for event E1, T2 runs.
t3	T2	the allocated time has expired (t1+2 ticks)	Allocated time has expired, but there is no eTS function call.
t4	T2	**sync_reschedule()**	eTaskSync returns control to the main loop because the allocated time has expired, T1 remains in the Wait queue, T2 remains in the Run queue.
t5	ISR	interrupt sets event: **set_event (E1)**	Interrupt service routine sets event E1.
t6	main loop	**sync_run (4)**	Main loop calls eTaskSync. T2 is moved to the Ready queue and T1 starts to run because event E1 has been set and T1 has higher priority than T2.



2.5 Mutexes

Mutex Overview

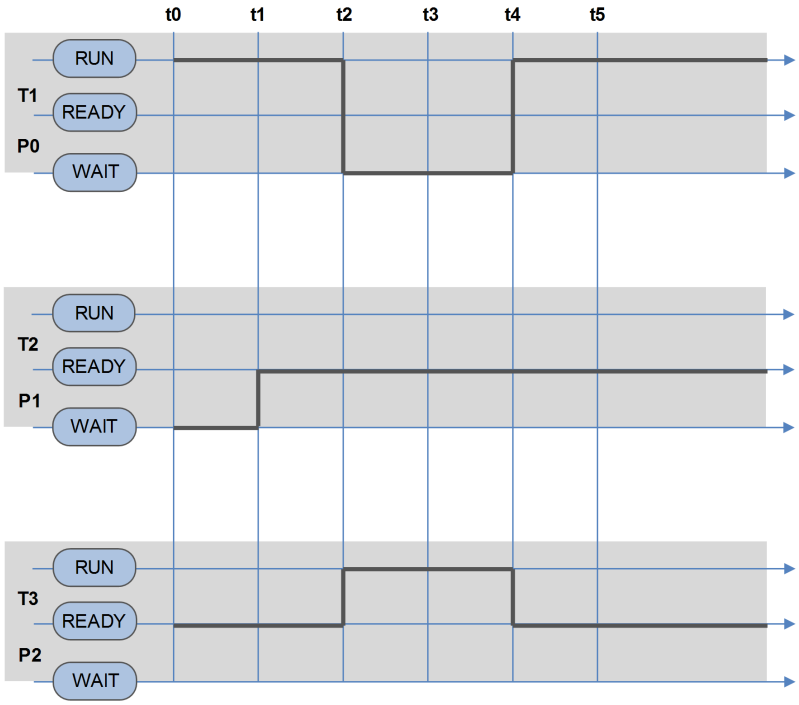
Mutexes are mutual exclusion locks. You use mutexes to guarantee that, while one task is using a particular resource, no other task can pre-empt it and use the same resource. Typically you will assign a particular mutex to protect a particular resource like a memory area or IO device. Every user of that resource should lock the mutex by using **sync_mutex_get()** before accessing it, and unlock the mutex by using **sync_mutex_put()** when it has finished with the resource. It is normally desirable to minimize the amount of time mutexes are locked for.

Mutex nesting is not supported. In other words you should not lock a mutex more than once from a task without first unlocking it. Locking multiple different mutexes from within a task is supported, though you must always take care to ensure that you do not create a deadlock situation with other tasks using the same mutexes. For example, if task A allocates mutex A then B, and task B allocates mutex B then A, a deadlock can result where neither task can proceed.

The time sequence diagram below illustrates how mutexes can be used between tasks. Here T1 has priority P0, T2 has P1, and T3 has P2. When working with mutexes it is also important to consider issues of priority inversion which are documented in Priority Inheritance.

There is no association between a particular task and a particular mutex, so it is possible for one task to lock a mutex and a different task to unlock it, though designing systems in this way is not recommended. It is your responsibility to manage the way a mutex is used to protect a resource.

Tick	Task	Action	Notes
t0	T1	Initial state	T1 is running and has locked mutex M2. T2 is waiting for mutex M2. T3 is ready to run and has locked mutex M1.
t1	T1	**sync_mutex_put(M2)**	Unlocks mutex M2 and reschedules. T2 is moved to the Ready queue. T1 continues running as it is the highest priority ready task.
t2	T1	**sync_mutex_get(M1)**	T1 waits for mutex M1, that T1 has locked. T3 runs at the priority level of T1 (priority inheritance).
t3	T3	**sync_task_reschedule()**	T3 continues running since its inherited priority is the highest among the ready tasks.
t4	T3	**sync_mutex_put(M1)**	T3 unlocks mutex M1 and reschedules. T1 gets mutex M1 that it was waiting on. T1 runs.
t5	T1	**sync_task_reschedule()**	T1 continues running.



Note: Unlike the case of a fully pre-emptive system, there is no need to protect a resource with a mutex if there are no calls to eTS functions during that access period. In general, we recommend that it is good practise to use a mutex; note that the actual overhead is minimal.

Priority Inheritance

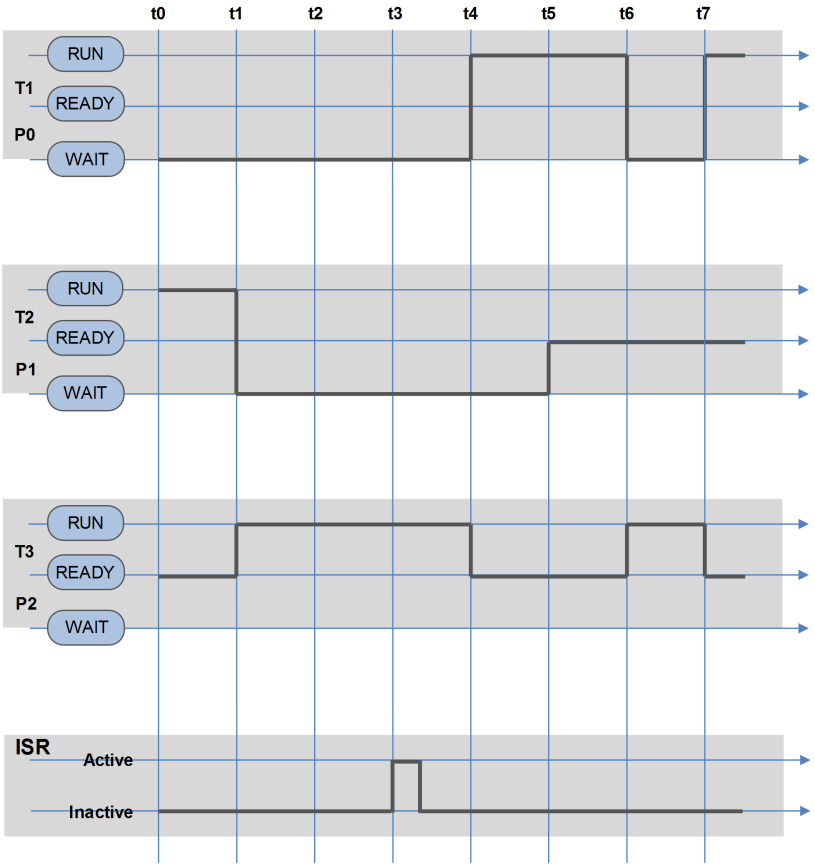
eTaskSync supports priority inheritance.

If a mutex is locked by a lower priority task than that which is requesting a mutex, the priority of that lower priority task is raised to the priority of the higher priority task until it has unlocked that mutex. This is termed priority inheritance.

The purpose of priority inheritance is to avoid a situation where a lower priority task blocks the progress of a higher priority task. This avoids the problem where a task of medium priority delays a lower priority task which is holding the mutex that the higher priority task requires. In this case the middle priority task effectively blocks a higher priority task. When the lower priority task has unlocked the mutex, it returns to its normal priority level, and the higher priority task can run.

The diagram below illustrates this process. (Here, T1 has priority P0, T2 has P1, and T3 has P2.)

Tick	Task	Action	Notes
t0		initial state	T1 is waiting for event E1. T2 is running. T3 is in the ready queue.
t1	T2	get_event(E2)	T2 waits for event E2. T3 runs.
t2	T3	**sync_mutex_get(M1)**	Mutex M1 is available. T3 gets mutex M1.
t3	ISR	set_event(E1)	Asynchronous event (ISR) sets the event E1.
t4	T3	**sync_task_reschedule()**	T1 gets event E1 and starts to run (P0 is higher priority than P2).
t5	T1	set_event(E2)	Event E2 has been set. T2 is ready to run. T1 keeps on running (P0 is higher than P1).
t6	T1	**sync_mutex_get(M1)**	T1 waits for mutex M1. T3 locks mutex M1. T3 inherits priority P0 and runs.
t7	T3	**sync_mutex_put(M1)**	T3 releases mutex M1, priority restore: T1-P0, T3-P2, T1 runs.



2.6 Usage

eTaskSync can be run either as the main scheduler of a system, or as a sub-system of another system.

As a main scheduler

Here **sync_run()** is called once and eTaskSync controls all the tasks based on their priority and other control requirements such as mutexes and events.

In a non RTOS or super loop system

Here eTaskSync runs as a sub-system of another system. Function **sync_run()** is called with a specified time to run for. While that time has not expired, eTaskSync schedules tasks based on their priorities. If all the tasks are in a waiting state, **sync_run()** returns before the time to run is complete.

This mode of operation is typical when a system has no RTOS and there is complex middleware to be scheduled. The super loop can call **sync_run()** and allow it to manage all the tasks required for a defined period of time, thus reducing the complexity of the super loop.

In an RTOS system

eTaskSync can be run as a task (maybe of very low priority) in an RTOS to isolate a particular set of functionality and execute this within a particular context or time frame, independent of the other activities of the system. In this environment, if the host system requires access to functions running within the eTS context, then a non-blocking API for those functions must be used. HCC has a standard mechanism for implementing non-blocking APIs for any module.

2.7 eTS Tick

All time measurements in eTS are based on the tick rate you provide by using the PSP function sync_get_tick_count().

Normally this tick count would be provided by the host system, using a timer or another provided mechanism. eTS does not care what the granularity of this timer is. The accuracy of the eTS timing will always be within -1 tick of the requested tick time.

Note: Because this is a co-operative scheduler, the current tick counter can only be checked when an eTS function is called. It is the responsibility of each task to ensure it meets the timing requirements of the system as a whole.

2.8 Real Time Characteristics

One of the big questions facing all schedulers and RTOSes is "what are the real time characteristics of the system?" In its simplest terms this equates to "what is the guaranteed worst case latency for a specified operation to take place?".

In a conventional pre-emptive RTOS:

- The worst case latency for the highest priority task is normally the worst case execution time of all the interrupts that can occur simultaneously. In this situation there can be only one highest priority task, otherwise the latency of all the tasks at that priority level must be added.
- The latency of a lower priority task needs to add the potential latency of all the higher priority tasks to obtain the total interrupt execution time. From a practical point of view, to achieve high levels of real-time behavior, the critical functionality must be placed either under interrupt control, or in a very high priority task.

Because eTaskSync is a co-operative scheduler, here things are subtly different. To the latency of the highest priority task, you must add the worst case time for any lower priority task to call an eTS function. This means that all tasks must be designed with the system requirements in mind. This is in contrast to a pre-emptive system, where all the tasks of lower priority than the time-critical tasks can be designed without regard to the system timing requirements.

3 Source File List

This section lists and describes all the source code files included in the system. These files follow HCC Embedded's standard source tree system, described in the *HCC Source Tree Guide*. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

> **Note:** Do not modify any files except the configuration file.

3.1 API File

The file **src/api/api_sync.h** must be included by any application using the system. It includes all the functions and declarations needed to initialize and run eTS. For details of the API functions, see Application Programming Interface.

3.2 Configuration File

The file **src/config/config_sync.h** contains all the configurable eTS system parameters. Configure these as required. For details of these options, see Configuration Options.

3.3 Source Code Files

These files are in the directory **src/sync**. **These files should only be modified by HCC.**

File	Description
sync_csw.h	Header file for eTS context switch code.
sync_debug.c	Source file for eTS debug code.
sync_event.c	Source file for eTS event code.
sync_event.h	Header file for eTS event code.
sync_mutex.c	Source file for eTS mutex code.
sync_sched.c	Source file for eTS scheduler code.
sync_sched.h	Header file for eTS scheduler code.
sync_task.c	Source file for eTS task code.
sync_task.h	Header file for eTS task code.

3.4 Version File

The file **src/version/ver_sync.h** contains the version number of this module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

3.5 Quality Metrics

These files contain the results of running static and dynamic coverage tests on eTS. The dynamic coverage tests, including full MC/DC tests, are performed using the eTS test suite.

File	Description
doc/sync/LDRA/coverage/*.*	Set of dynamic code coverage test results generated using the eTaskSync test suite.
doc/sync/LDRA/review/*.*	Set of static analysis results including comprehensive MISRA checking for eTS code.

4 Configuration Options

Set the system configuration options in the file **/src/config/config_sync.h**.

SYNC_RUN_FOREVER

If this is enabled, **sync_run()** has no parameter passed to it and executes forever. This is the same as executing the main loop of a program or an RTOS.

If this is disabled, the number of ticks it can run is passed to it as a parameter. The default value is zero.

SYNC_TIME_SLICE_ENABLE

This enables time slicing. The default value is 1.

SYNC_DEBUG_ENABLE

This enables the debug module. It provides the option to get stack usage information and the basic level of processor usage of eTS Tasks. The default value is zero.

5 Application Programming Interface

This section documents the Application Programming Interface (API). It includes all the functions that are available to an application program.

5.1 System Functions

This section describes the following functions that are used to start and run eTaskSync.

Function	Description
sync_init()	Initializes the eTaskSync module.
sync_run()	Executes tasks within eTaskSync for the period specified.
sync_run() - forever	Executes tasks within eTaskSync forever.

sync_init

Use this function to initialize the eTaskSync module.

> **Note:** Call this once before performing any other eTS operations.

Format

```
t_sync_ret sync_init ( void )
```

Arguments

Argument
None.

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

sync_run

Use this function to execute tasks within eTaskSync for the period specified.

eTS can only return when tasks within eTS call eTS functions. When the time has expired, this function returns to the caller.

Format

```
void sync_run ( uint32_t ticks )
```

Arguments

Argument	Description	Type
ticks	The maximum number of ticks eTaskSync can run for.	uint32_t

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

sync_run - forever

Use this function to execute tasks within eTaskSync forever.

Use this version of the **sync_run()** function if the SYNC_RUN_FOREVER configuration option is enabled.

Format

```
void sync_run ( void )
```

Arguments

Argument
None.

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

5.2 Task Functions

This section describes the following task functions:

Function	Description
sync_task_create()	Creates a task.
sync_task_delete()	Deletes a task.
sync_task_get_id()	Gets the ID of the running task.
sync_task_reschedule()	Forces eTS to schedule a task.
sync_task_sleep()	Suspends the running task for the specified number of ticks.
sync_task_yield()	Allows tasks with lower priority than the calling task to run.

sync_task_create

Use this function to create a task. The new task is immediately added to the Ready queue.

The caller must do the following:

- Allocate a task structure, *t_sync_task*, and pass its pointer to this function.
- Create a task descriptor as described below, and pass its pointer to this function.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	Yes	No

Format

```
t_sync_ret sync_task_create (
    t_sync_task * const             p_task,
    t_sync_task_dsc const * const   p_task_dsc,
    uint16_t * const                p_task_id )
```

Arguments

Argument	Description	Type
p_task	A pointer to the task structure.	t_sync_task *
p_task_dsc	A pointer to the task descriptor.	t_sync_task_dsc *
p_task_id	A pointer for writing the ID of the created task.	uint16_t *

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

Task descriptor

The caller must set parameters for the task to be created.

The task descriptor is as follows:

```
typedef void ( *t_task_entry )( void );

    typedef struct
    {
        char *          p_name;         /* pointer to the task's name */
        t_task_entry    entry;          /* entry point of the task */
        void *          p_stack_ptr;    /* pointer to the stack */
        uint32_t        stack_size;     /* size of the stack */
        uint8_t         pri;            /* priority of the task */
    #if SYNC_TIME_SLICE_ENABLE
        uint32_t        time_slice;     /* timeslice ticks (if enabled) */
    #endif
    } t_sync_task_dsc;
```

Field	Description
p_name	A pointer to a zero-terminated character string containing the name of the task.
entry	The entry point to the task. That is, the main function of the task.
p_stack_ptr	A pointer to the memory block to be used as a stack for this task. eTS will align this to four bytes.
stack_size	The size of the stack that has been allocated to this task.
pri	The priority of this task. Zero is the highest, 254 is the lowest.
time_slice	The time for which this task can execute before other tasks of the same priority are scheduled.

sync_task_delete

Use this function to delete a task.

The running task can delete itself or another task.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	Yes	No

Format

```
t_sync_ret sync_task_delete ( t_sync_task * const p_d_task )
```

Arguments

Argument	Description	Type
p_d_task	A pointer to the task structure of the task to be deleted.	t_sync_task *

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

sync_task_get_id

Use this function to get the ID of the running task.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	No	No

Format

```
uint16_t sync_task_get_id ( void )
```

Arguments

Argument
None.

Return Values

Return value	Explanation
Task ID.	Successful execution.
Else	See Error Codes.

sync_task_reschedule

Use this function to force eTS to schedule a task.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	No	No

Format

```
void sync_task_reschedule ( void )
```

Arguments

Argument
None.

Return Values

Return value
None.

sync_task_sleep

Use this function to suspend the running task for the specified number of ticks.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	No	No

Format

```
void sync_task_sleep ( uint32_t ticks )
```

Arguments

Argument	Description	Type
ticks	The number of ticks to sleep for.	uint32_t

Return Values

Return value
None.

sync_task_yield

Use this function to allow tasks with lower priority than the calling task to run.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	No	No

Format

```
void sync_task_yield ( void )
```

Arguments

Argument
None.

Return Values

Return value
None.

5.3 Event Functions

This section describes the following event functions:

Function	Description
sync_event_init()	Initializes an event group.
sync_event_delete()	Deletes an event group.
sync_event_flags_clear()	Clears flag(s) in an event group.
sync_event_flags_get()	Waits for one or more of a specified set of flags in an event group to be set.
sync_event_flags_set()	Sets flag(s) in an event group.
sync_event_flags_set_async()	Sets flag(s) in an event group asynchronously.

sync_event_init

Use this function to initialize an event group. This must be called once before any other event call is made on that event group.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	Yes	No

Format

```
t_sync_ret sync_event_init ( t_sync_event * const p_event )
```

Arguments

Argument	Description	Type
p_event	A pointer to the event group.	t_sync_event *

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

sync_event_delete

Use this function to delete an event group.

If a task is waiting for this event, a SYNC_ERR_RESOURCE is returned by **sync_event_flags_get()**.

After deletion, you must call **sync_event_init()** before a deleted event group can be used again.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	Yes	No

Format

```
t_sync_ret sync_event_delete ( t_sync_event * const p_event )
```

Arguments

Argument	Description	Type
p_event	A pointer to the event group.	t_sync_event *

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

sync_event_flags_clear

Use this function to clear flag(s) in an event group.

This function clears only those flags that have not yet been delivered; that is, where no task was waiting for a flag when it was set, or a flag has been set asynchronously and has not yet been processed by the scheduler.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	No	No

Format

```
t_sync_ret sync_event_flags_clear (
    t_sync_event * const    p_event,
    uint32_t                c_flags )
```

Arguments

Argument	Description	Type
p_event	A pointer to the event group.	t_sync_event *
c_flags	The flag(s) to clear.	uint32_t

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

sync_event_flags_get

Use this function to wait for one or more of a specified set of flags in an event group to be set. Any flags that are set are cleared when this function returns.

You can also specify a maximum number of ticks to wait if none of the specified flags has been set. If the maximum number of ticks is exceeded, the function returns a SYNC_ERR_TIMEOUT error.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	No	No

Format

```
t_sync_ret sync_event_flags_get (
    t_sync_event * const    p_event,
    uint32_t                w_flags,
    uint32_t * const        p_s_flags,
    uint32_t                timeout )
```

Arguments

Argument	Description	Type
p_event	A pointer to the event group.	t_sync_event *
w_flags	The flag(s) to wait for.	uint32_t
p_s_flags	Where to write the obtained event flags.	uint32_t *
timeout	One of the following: • The number of ticks to wait before timeout return. • SYNC_WAIT_FOREVER to wait indefinitely.	uint32_t

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

sync_event_flags_set

Use this function to set flag(s) in an event group.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	No	No

Format

```
t_sync_ret sync_event_flags_set (
    t_sync_event * const   p_event,
    uint32_t               s_flags )
```

Arguments

Argument	Description	Type
p_event	A pointer to the event group.	t_sync_event *
s_flags	The flag(s) to set.	uint32_t

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

sync_event_flags_set_async

Use this function to set flag(s) in an event group asynchronously.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	Yes	Yes

Note: Although this can be called from the eTS context, using **sync_event_flags_set()** is always better. The only reason to use this asynchronous function from the eTS context is to set an event flag or flags without allowing the scheduler to run.

Format

```
t_sync_ret sync_event_flags_set_async (
    t_sync_event * const   p_event,
    uint32_t               s_flags )
```

Arguments

Argument	Description	Type
p_event	A pointer to the event group.	t_sync_event *
s_flags	The flag(s) to set.	uint32_t

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes .

5.4 Mutex Functions

This section describes the following mutex functions:

Function	Description
sync_mutex_init()	Initializes a mutex
sync_mutex_delete()	Deletes a mutex.
sync_mutex_get()	Locks a mutex.
sync_mutex_put()	Unlocks a mutex.

sync_mutex_init

Use this function to initialize a mutex.

All mutexes must be initialized before use.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	Yes	No

Format

```
t_sync_ret sync_mutex_init ( t_sync_mutex * const p_mutex )
```

Arguments

Argument	Description	Type
p_mutex	A pointer to the mutex.	t_sync_mutex *

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

sync_mutex_delete

Use this function to delete a mutex.

If a task is waiting for this mutex, a SYNC_ERR_RESOURCE is returned by **sync_mutex_get()**.

After deletion, you must call **sync_mutex_init()** before the mutex can be used again.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	Yes	No

Format

```
t_sync_ret sync_mutex_delete ( t_sync_mutex * const p_mutex )
```

Arguments

Argument	Description	Type
p_mutex	A pointer to the mutex.	t_sync_mutex *

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

sync_mutex_get

Use this function to lock the specified mutex.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	No	No

Format

```
t_sync_ret sync_mutex_get ( t_sync_mutex * const p_mutex )
```

Arguments

Argument	Description	Type
p_mutex	A pointer to the mutex.	t_sync_mutex *

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

sync_mutex_put

Use this function to unlock the specified mutex.

> **Note:** Only call this to release a previously locked mutex.

Context

eTaskSync	Non-Interrupt	Interrupt
Yes	No	No

Format

```
t_sync_ret sync_mutex_put ( t_sync_mutex * const p_mutex )
```

Arguments

Argument	Description	Type
p_mutex	A pointer to the mutex.	t_sync_mutex *

Return Values

Return value	Explanation
SYNC_SUCCESS	Successful execution.
Else	See Error Codes.

5.5 Error Codes

All functions in eTaskSync are either void (they have no return value), or return with the error code in t_sync_ret.

The possible return codes are shown in the table below:

Code	Value	Description
SYNC_SUCCESS	0	Function successfully executed.
SYNC_ERR_RESOURCE	1	The requested resource could not be initialized.
SYNC_ERR_TIMEOUT	2	This is generated by **sync_event_flags_get()** if it times out.
SYNC_ERR_INITIALIZED	3	The requested resource is already initialized.
SYNC_ERR_INVALID_PRI	4	The requested priority level is out of range.

6 Debug Module

A debug module is provided to help you with task stack analysis and task performance analysis.

This module is enabled by using the SYNC_DEBUG_ENABLE configuration option.

> **Note:** This code is for development use only. For this reason it does not meet the same coding standard as the release code.

The module has three functions, described in the following sections.

6.1 sync_debug_pf_start

Use this function to start the performance test program.

Format

```
void sync_debug_pf_start ( void )
```

Arguments

Argument
None.

Return Values

Return value
None.

6.2 sync_debug_pf_stop

Use this function to output a printable message of the performance measurement since the last call to **sync_debug_pf_start()**.

The output data gives:

- The number of ticks during the measurement.
- The ticks of idle time during the measurement.
- The load – a measure of the time spent executing the task as a percentage of the total time available.

> **Note:** The test is designed to run where the system is dedicated to running eTS only. Only in this situation will the report be accurate.

Format

```
void sync_debug_pf_stop (
    char *      buf,
    uint32_t    buf_len )
```

Arguments

Argument	Description	Type
buf	A pointer to the buffer to write the output data to.	char *
buf_len	The size of the above buffer.	uint32_t

Return Values

Return value
None.

6.3 sync_debug_task_info

Use this function to output detailed data about the state of the task.

The function builds a printable message about the task. This includes its name, its ID, the total stack size, and the amount of stack used.

Format

```
void sync_debug_task_info (
    char *        buf,
    uint32_t    buf_len )
```

Arguments

Argument	Description	Type
buf	A pointer to the buffer to write the output data to.	char *
buf_len	The size of the above buffer.	uint32_t

Return Values

Return value
None.

7 Integration

This section describes all aspects of the module that require integration with your target project. This includes porting and configuration of external resources.

eTaskSync must be integrated with the target environment. To do this, you must make certain functions that can only be implemented specific to the target system available to it. Generally these functions are included in the Platform Support Package (PSP) supplied by HCC, but they may need modification to match the target requirements.

Two sets of functions are described in the sections below:

- **Architecture-specific** – for executing a context switch between tasks. These functions are normally provided by HCC, but this section describes them to enable use of eTaskSync on an architecture that is not supported by HCC.
- **Platform-specific** – these relate to a particular project. These porting functions are for the provision of a tick counter and enable/disable interrupts from the host system.

7.1 Architecture-specific Functions

For each target micro-controller architecture, some functions that are specific to that controller must be provided to perform a context switch for eTaskSync. Normally this is provided by HCC as one of the add-on packages specific to a target and a toolchain. This section details the functions that must be provided by each of these packages.

Note: These functions are only of interest to developers porting eTS to a new target.

Note: One PSP type definition must be set up when porting eTS to a new architecture: t_sync_sp in **psp_sync_sp.h**. This is the size of the stack pointer and this must be set to a size equal to or greater than the size of the stack pointer on the target system.

On most micro-controllers this can be set to unsigned int since the stack pointer and register sizes are normally equal, but this is not always the case. For example, on some MSP430 micro-controllers in some memory models the stack pointer is 20 bits and, in this case, it would be normal to set this typedef to a 32 bit quantity.

sync_context_init

Use this function to initialize the context.

This is needed:

- To initialize the stack pointer according to the target.
- To push all registers and the entry point to the stack, in order to enter the task using **sync_context_switch()**.

Format

```
void sync_context_init (
    t_sync_sp *     p_task_sp,
    uint32_t        s_size,
    t_task_entry    t_entry )
```

Arguments

Argument	Description	Type
p_task_sp	A pointer to the stack pointer of the task to initialize. By default this points to the start of the stack, so the function must set this (using *s_size*) according to the system.	t_sync_sp *
s_size	The stack size in uint32_t units.	uint32_t
t_entry	The entry point of the task.	t_task_entry

Return Values

Return value
None.

sync_context_switch

Use this function to perform a context switch.

This function does the following:

1. Saves all the registers and the Program Counter (PC) of the old task to the current stack, then writes the stack pointer to *p_otask_sp*.
2. Loads the stack pointer from *p_ntask_sp*, then restores the registers and the Program Counter of the new task.

Format

```
void sync_context_switch (
    t_sync_sp *   p_otask_sp,
    t_sync_sp *   p_ntask_sp )
```

Aguments

Argument	Description	Type
p_otask_sp	A pointer to the old task's stack pointer.	t_sync_sp *
p_ntask_sp	A pointer to the new task's stack pointer.	t_sync_sp *

Return Values

Return value
None.

7.2 Platform-specific Functions

sync_tick_init

Use this function to initialize the tick counter module.

If eTaskSync is used without an RTOS, this function can initialize a timer module. When it is used with an RTOS, this initialization is normally performed by the host system, in which case this function is not needed.

Format

```
void sync_tick_init ( void )
```

Arguments

Argument
None.

Return Values

Return value
None.

sync_get_tick_count

Use this function to return the current tick count value.

Format

```
t_sync_tick sync_get_tick_count ( void )
```

Arguments

Argument
None.

Return Values

Return value
The current tick count.

sync_int_disable

Use this function to disable global interrupts.

The return value, the original state of the interrupts, is the input parameter of **sync_int_restore()**.

Format

```
t_sync_int_save sync_int_disable ( void )
```

Arguments

Argument
None.

Return Values

Return value
Original state of the interrupts.

sync_int_restore

Use this function to restore interrupts to a previous state.

Always call **sync_int_restore()** after **sync_int_disable()**. This should restore the state of the interrupts to that prior to a **sync_int_disable()** call.

Format

```
void sync_int_restore ( t_sync_int_save int_save )
```

Arguments

Argument	Description	Type
int_save	The state of the interrupts.	t_sync_int_save

Return Values

Return value
None.

7.3 PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer.

The eTaskSync module makes use of the following standard PSP functions:

Function	Package	Component	Description
psp_strncat()	psp_base	psp_string	Appends a string.
psp_strncpy()	psp_base	psp_string	Copies one string of defined length to another.
psp_strnlen()	psp_base	psp_string	Gets the length of a string.