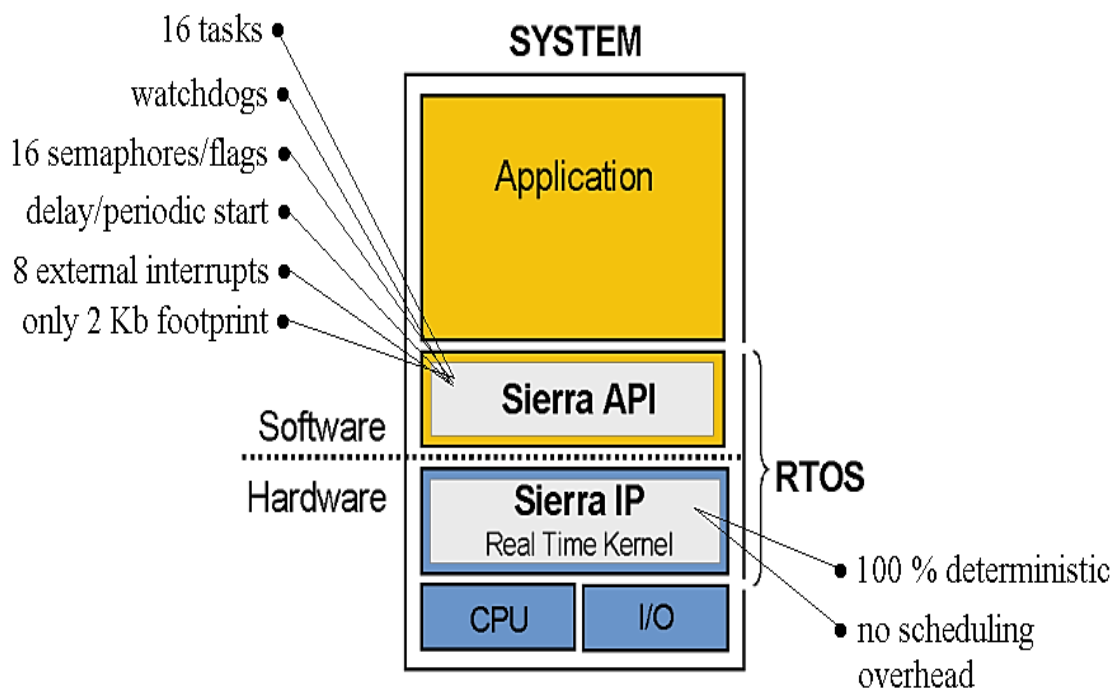


Sierra



This manual covers the use of Sierra RTK, all functions may not be implemented in the version you chose. Configuration and some implementation results of the different Sierra, see web page www.agstu.com. The educational Sierra have not implemented all the functions described in this documentation.

© Copyright by publisher AGSTU AB.
www.agstu.com

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

The author and publisher of this manual make no warranty of any kind.

Contents

Introduction	5
About This Manual	5
Revision History	5
Purpose	6
Terms	6
Sierra Overview	7
Sierra HW	7
HW Scheduler	8
Setup and Initiating	10
Sierra Hardware/Software Initiation	10
Description	10
Function declaration	10
Argument	10
Return codes	10
Set and Read Time Base Register	10
Description	10
Function declaration	11
Argument	11
Return codes	11
Example	11
Low level API	12
Task Management	12
sierra_create_task	12
sierra_start_task	13
sierra_get_task_info	14
sierra_tsw_off	15
sierra_tsw_on	16
sierra_block_task	16
sierra_change_task_prio	17
sierra_delete_task	17
IRQ Management	18
sierra_wait_irq	19
Time Management	19
sierra_period_time_init	21
sierra_wait_next_period	21
Semaphore Management	23
sierra_take_sem	23
sierra_release_sem	24
sierra_read_sem	25
Flag Management	25
sierra_wait_flag	27
sierra_set_flag	27
sierra_clear_flag	28
Sierra information calls	29

Sierra HW Version.....	29
Sierra SW Version	30
Logging API	32
Sierra Time Logging Register	32
ON/OFF Logging System	32
Logging functions	32
Sierra Logging Probes.....	33
Extended API	35
Mailbox	35
sierra_mbox_get_required_size	35
sierra_mbox_init	36
sierra_mbox_send	36
sierra_mbox_read and sierra_mbox_peak.....	37
Hardware interface	39
Protocol with external start of blocked task (extended Sierra).....	40
Sierra SW File Structure	41

Introduction

About This Manual

This manual is describing the low-level API service calls direct to the hardware-based Sierra.

The Sierra RTK/RTOS consists of following parts (see next figure):

1. Hardware based Sierra is connected to the data bus, it works as a hardware accelerator for real-time kernels/operating systems
2. Software_Reference_Manual is a description of the basic RTK device drivers to Sierra HW

Revision History

Date	Description
2013-03-18	Updated the documentation
2014-07-18	Updated the documentation
2015-02-03	Added task delete and some text debugging
2016-03-03	Added "task_change_prio" and some text debugging, version 9.2
2016-04-17 v 9.3.1	Change in the scheduler; lowest priority is 0. Same as FreeRTOS Add Block task of other then the running. Same as FreeRTOS Update version register
2016-05-01 V 9.4.0	#semaphore is not bounded to #tasks Sierra Version register updated (see Sierra HW Version)
2017-10-29 V9.4.1	Some updates and optimizations, also a new students Sierra.
2020-04-01	New crypted version of HW Sierra
2022-09-27 V 10.03.15	<ul style="list-style-type: none">- Added "sierra_print_versions()",- Added prefix sierra_ for functions,- Updated documentation.- Add a logging timer, count on the time ticks (32 bits). CPU Time Logging Register, working with Sierra HW version 9.5.0 and higher.- Update SW test and add test_000_testing_time_register- Some errors in test cases fixed- Logging functions with probes in Sierra- Extended with Mailbox Change task test is not working in Sierra_HW version 9.5.0 (bug)

Purpose

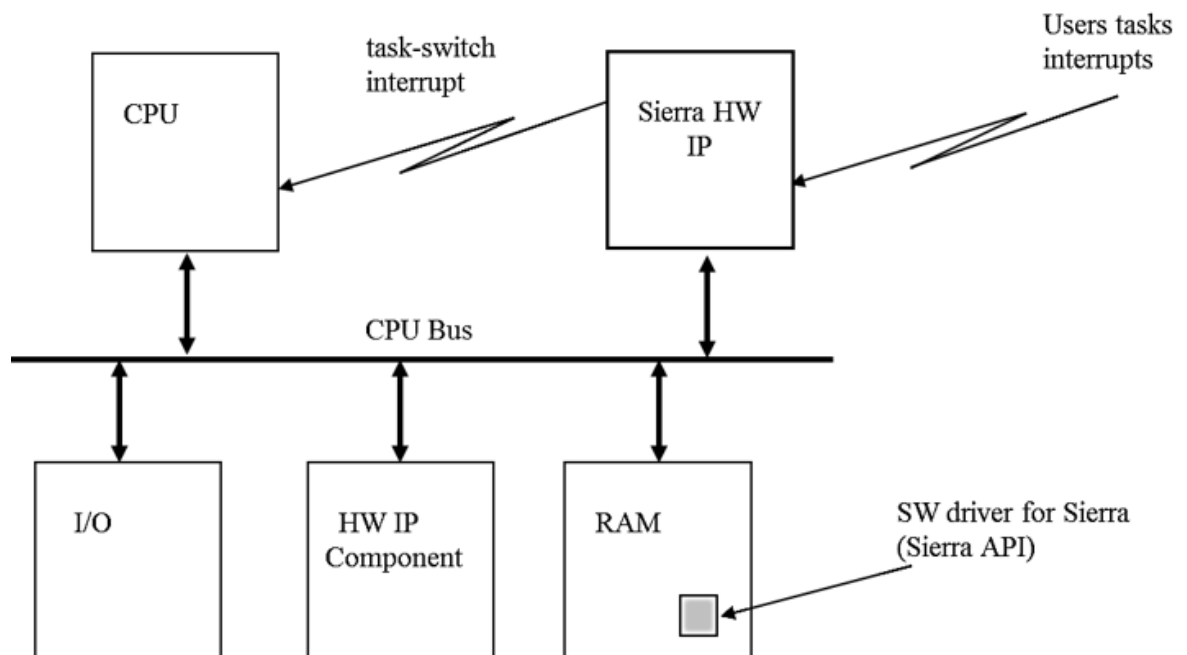
The purpose of this Reference Manual is to give programmers the API to Sierra Real-Time Kernel.

Terms

API	Application Programmers Interface, The sum of all function calls available to an application programmer
Application mode	A description of a complete system with scheduler, tasks etc. some RTOS allows the programmer to specify more than one mode. I.e., an aircraft control system may have different modes for takeoff, landing, and level flight.
Context switch (task switch)	Switch from current running task to another task by saving current task status, registers etc., and restore status of the task that shall start to run.
Embedded system	A computer system that forms a component of a larger system and is expected to function without human intervention.
Exception	Software interrupts.
Interrupt service routine (ISR)	The routine that is called when an interrupt occurs.
IP	Intellectual Property, this is HW/SW components with a specific function.
Real-time system	A real-time system is one in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are generated.
RTOS	Real time operating system, an operating system designed to be used in real time systems.
Task/Thread/Process	A task is a sequential programming performing certain functions, a real time application is usually made up of one or more sets of communicating tasks.
TCB	Task control block, a structure containing information about a task, its state, stack owned resources, the value of the processor registers etc.

Sierra Overview

Ordinary real-time operating systems are implemented in software. The Sierra are the basic functions implement in hardware, i.e., scheduling, inter process communication, interrupt management, tick handling, logging, and time management control into hardware. This makes it possible to take advantage of hardware characteristics such as parallelism and robustness that consequently decreases system overhead and decrease response time.



The figure shows Sierra implemented in a computer architecture. The Sierra HW IP is connected to the CPU bus, driver interface, in software (Sierra API) and an interrupt connection to the CPU. The task-switch interrupt to the CPU starts pre-emption of the running task.

Sierra HW

The Sierra HW core is partitioned into modules as shown in the figure.

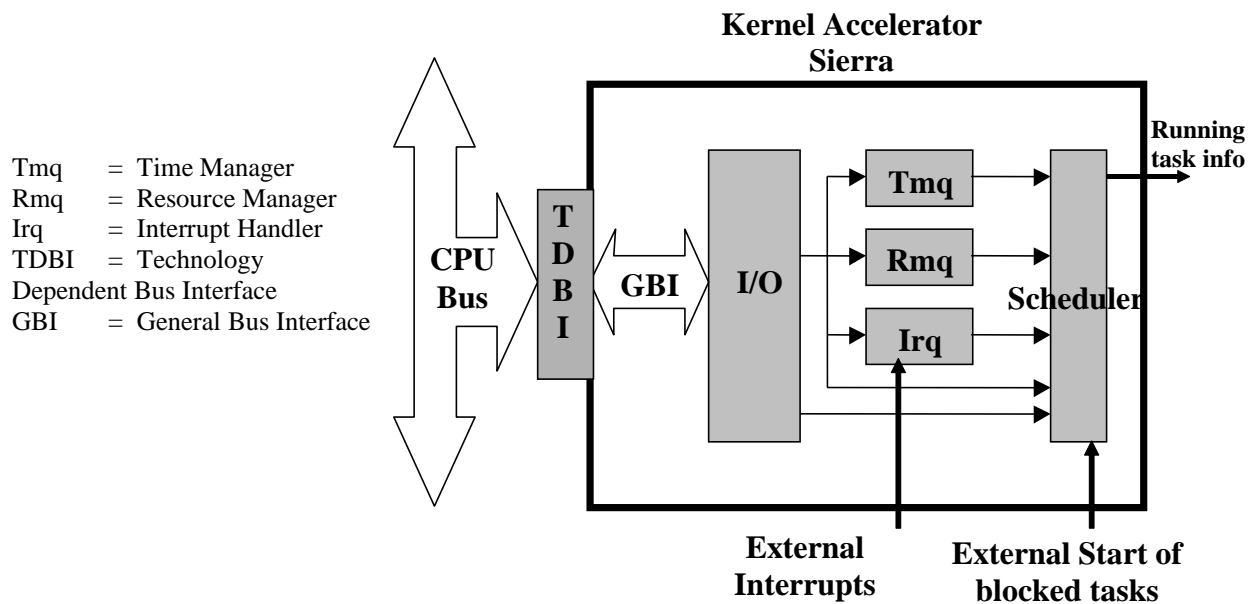


Figure 1. Overview of internal blocks in Sierra HW part

Sierra RTOS is partitioned into these functional units:

Interface
 Scheduler
 Semaphore and Flag Handler
 Time Management Controller
 Tick counter for logging

The interface to Sierra is divided into a generic bus interface (GBI) and a technology dependent bus interface (TDBI). The GBI is bus independent while the TDBI is glue to the specific bus in the system. This design of the Sierra makes it very easy to interface it towards different busses.

All communication (service calls) with the Sierra is carried out through registers. In the internal module interface, the service calls are decoded and routed out the unit that will carry out the service call. This interface synchronizes external accesses from the CPU as well as all internal work between modules in the chip.

The Sierra hardware can be configured as following:

- 4-512 tasks
- 4-512 priority levels
- 4 -1024 semaphores
- 4 -1024 flags
- 4 – 512 Timers for delay, periodic tasks
- 2 – infinite interrupts

More about the hardware in the book “Advanced HW/SW Embedded System for Designer” (Amazon)

HW Scheduler

The Scheduler unit controls all task scheduling in the Sierra. The scheduler can handle tasks at different priority levels. Tasks can also be created and deleted dynamically during runtime. When a task is created, it is initialized to a specified state (blocked or ready).

A task can exist in five different states; running, ready, blocked, waiting for IRQ or dormant. The scheduler guarantees that the task with highest priority among the ready tasks always will run.

The Sierra can support the following task states and transitions:

- Running
- Ready
- Blocked /Waiting
- Wait for interrupt
- Dormant

Setup and Initiating

Sierra Hardware/Software Initiation

Description

Initiate the TCB in soft/hardware and resets the Sierra hardware. After initiation the task switch is off.

Function declaration

```
void sierra_initiation_HW_and_SW(void)
```

Argument

Nothing

Return codes

N/A

Set and Read Time Base Register

Description

Sets or read the internal clock-tick timebase for the Sierra. This register is used to set-up the generating of Sierra internal clock tick period for all timing queues in Sierra.

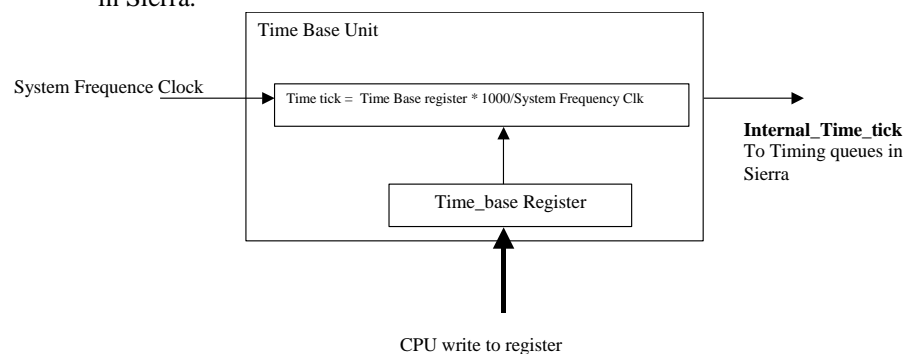


Figure 2. Time Base Unit

Sierra Time Base register value = Time tick * system Frequency/1000

Function declaration

set timebase	void sierra_set_timebase (unsigned int hex)
read timebase	unsigned int sierra_time_base_reg(void)

Argument

set timebase	See Sierra specification for number of bits:/13 bits; hex: range 0-8191, please check the version of hardware. unsigned int
read timebase	N/A

Return codes

set timebase	N/A
read timebase	unsigned int

Example

```
void t1(void)
{
    sierra_set_timebase(50); /* Set Sierra internal
                             clock-tick to 1ms
                             when the HW kernel
                             runs at 50 MHz
                             system clock*/
}
```

Low level API

Task Management

This section describes the task handling services provided by the scheduler in the Sierra. The difference between Sierra and other RTOS kernels is that all scheduling is performed by a hardware piece instead of software. The only software is the driver that communicates with the hardware kernel. The following task management functions are implemented in the Sierra hardware kernel:

- Dynamic creation of tasks (`sierra_create_task`)
- Starting of tasks (`sierra_block_task`)
- Yield (`sierra_yield_task`)
- Get task status (`sierra_get_task_info`)
- Task switch off and on (`sierra_tsw_on` and `sierra_tsw_off`)
- Change task priority

Ready que is organized in two ways (scheduling algorithm):

- Priority driven (lowest priority is 0)
- Same priority is sorted in ID number order, from low to high.
- Preemption

Idle task must be created with task ID 0 and lowest priority (0).

`sierra_create_task`

Description

Creates a task with a unique task id. The task will be initialized to a state (blocked or ready) as specified in the argument. It is possible to create new tasks dynamically during system execution. Idle task must be created and have task ID 0 and lowest priority (0).

Function declaration

```
void sierra_create_task (int taskID,  
                        int priority,  
                        int taskstate,  
                        void (*taskptr) (void),  
                        void *stackptr,  
                        int stacksz);
```

Argument

task ID Specifies the ID of the task (range depend on the version of Sierra).

An idle task must be created, and this task shall have task ID 0.

priority	Specifies the priority of the task. The range is dependent on the version), where 0 is the highest-priority level. Highest ID number is reserved only for the idle task.
taskstate	0 = task is initialized to the blocked state (BLOCKED_TASK_STATE) 1 = task is initialized to the ready state (READY_TASK_STATE)
taskptr	Pointer to code start for the task
stackptr	Pointer to task stack
stacksz	Size of the stack

Return codes

N/A

Example

```
#define IDEAL 0
#define READY 1
#define PRIO1 0
#define STACK1_SZ 200
#define T1 1
#define READY 1
#define PRIO1 1
#define STACK1_SZ 200

char stack1[STACK1_SZ];

void t1(void)
{
    task code;
}

void function(void)
{
    sierra_create_task(T1, PRIO1, READY, t1, stack1,
STACK1_SZ);
}
```

sierra_start_task

Description

Starts a task that is currently placed in blocked state (un-block the task). Starting a task means that the task is sent into the ready state (see section 2.4., Scheduler) and does not mean that the task starts to execute immediately. The task will be moved from blocked state to ready state.

Function declaration

```
void sierra_start_task (int taskId)
```

Argument

task ID Specifies the ID of the task (range depend on the version of Sierra).

Return codes

N/A

Example

```
#define T2 2

void t1(void)
{
    sierra_start_task(T2); /* t1 starts T2 */

    while(1)
    {
        /* Insert code*/
    }
}
```

sierra_get_task_info

Description

Get status information about a specified task.

Function declaration

```
task_info_t sierra_get_task_info (int taskid)
```

Argument

task ID Specifies the ID of the task (range depend on the version of Sierra)

Return codes

task_info_t	state_info (2 bits):
	0=Running
	1=Blocked
	2=Ready
	3=Dormant

priority (3 bits, depend on the version of Sierra),
7 is the lowest priority level and 0 is the highest.

Example

```
task_info_t info;

printf("Task1\n");
info = task_getinfo(Task1);
printf("  info.state_info = %d\n", info.state_info);
printf("  info.priority = %d\n", info.priority);
```

Return:

```
Task 1
    info.state_info = 2
    info.priority = 1
```

Special print function (sierra_info.c)

```
void sierra_task_info(void)
```

Return:

```
Idle
    info.state_info = 2
    info.priority = 0
Task 1
    info.state_info = 2
    info.priority = 1
Task 2
    info.state_info = 2
    info.priority = 2
Task 3
    info.state_info = 2
    info.priority = 3
```

sierra_tsw_off

Description

Disables task-switch interrupts in the system. This is useful when a critical section is entered. Anyhow, this call should be used with restrictions in a real time system as it has effects on how/when tasks can start to run. If this call is used, try to have the task-switch interrupt off as short time as possible.

Function declaration

```
void sierra_tsw_off (void)
```

Argument

N/A

Return codes

N/A

Example

```
void t1(void)
{
    while(1)
    {
        sierra_tsw_off(); /* Entering critical
                           section, turn off
                           task-switch interrupts */
    }
}
```

sierra_tsw_on

Description

Enables task-switch interrupt.

Function declaration

```
void sierra_tsw_on(void)
```

Argument

N/A

Return codes

N/A

Example

```
void t1(void)
{
    while(1)
    {
        tsw_on(); /* Leaving critical section - Turn on
                   task-switch interrupts */
    }
}
```

sierra_block_task

Description

Blocks the currently running task. The task will be moved from running state into blocked state. It is **not allowed** to block idle task.

Function declaration

```
void sierra_block_task (int taskId)
```

Argument

task ID	Specifies the ID of the task (range depend on the version of Sierra).
---------	---

Return codes

N/A

Example

```
#define T2 2

void t1(void)
{
    int i=0;
    while(1)
    {
        i++;
        if(i==10) /* Block t2 when I == 10 */
        {
            task_block(T2); i = 0;
        }
    }
}
```



```
}  
}
```

sierra_change_task_prio

Description

This call changes a task's priority to a specified priority. It is **not allowed** to change idle task priority.

Function declaration

```
void sierra_change_task_prio (int taskID,  
                             int priority);
```

Argument

- | | |
|----------|---|
| task ID | Specifies the ID of the task (range depend on the version of Sierra). |
| priority | Specifies the priority of the task. The range is dependent on the version), where 0 is the highest-priority level. Highest ID number is reserved for the idle task. |

Return codes

N/A

Example

```
#define T2 2  
#prio_5 5  
  
void t1(void)  
{  
  
    sierra_change_task_prio(T2,prio_5);  
    /* Task T2 gets priority 5 */  
  
    while(1)  
    {  
        /* Insert code here */  
    }  
}
```

sierra_delete_task

Description

Delete the running task, preformed from current executing tasks code. The task will be moved from the system and the task ID number will be free for use again. To restore a deleted task the removed task must be created again. It is **not allowed** to perform this call from the idle task.

Function declaration

```
void sierra_delete_task(void)
```

Argument

N/A

Return codes

N/A

Example

```
void t1(void)
{
    int i=0;
    while(1)
    {
        i++;
        if(i==10)
        {
            task_delete();
            i = 0;
        }
    }
    /* Removed t1 from the system when i==10 */
}
```

IRQ Management

This section describes the functionality of the Interrupt Manager. The interrupts are associated with an interrupt task, which is scheduled as an ordinary task in the system. External interrupt is connected to Sierras external IRQ pins. Each IRQ input is level sensitivity and active-high.

The following functions is implemented in hardware:

- Wait for interrupt

If several external interrupts occur simultaneously, the task associated with highest interrupt pins will be the first one sent to the ready queue.

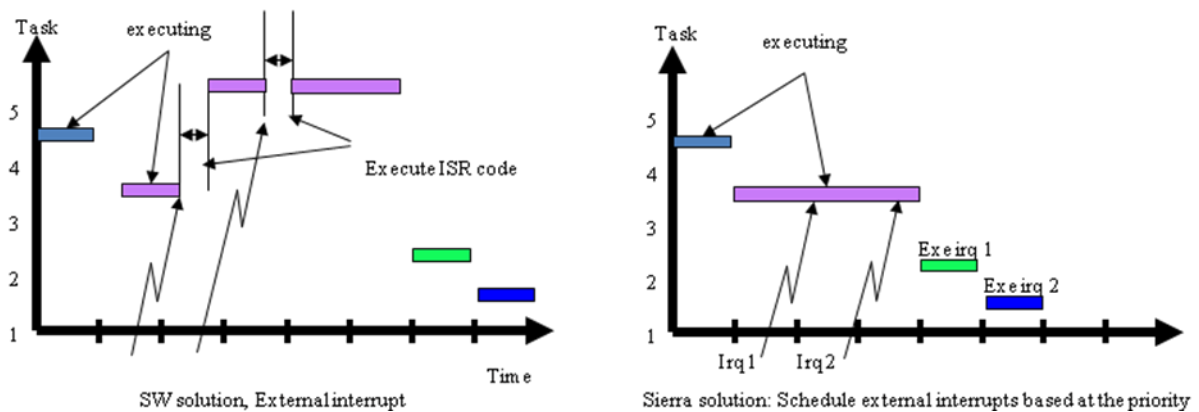


Figure 3. SW RTK and HW based Sierra solution, two low priority IRQ.

sierra_await_irq

Description

This call used when an interrupt service task is ready to process to wait for an external interrupt. As a result of this call, the interrupt service task (running task) will be moved to the 'Wait for interrupt' state.

The ID of task that the CPU should context switch too is in the return data.

Function declaration

```
void sierra_await_irq(int IRQ_number);
```

Argument

IRQ number Specifies the interrupt level. The range of the interrupt level depends on the version of the Sierra.

Return codes

N/A

Example

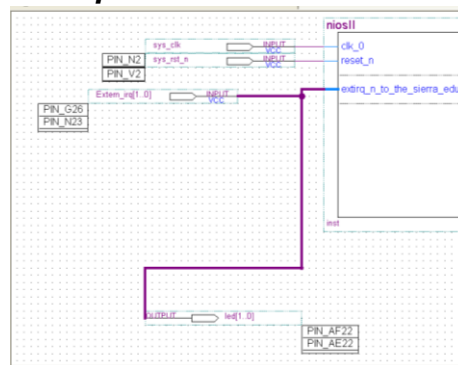


Figure 4. Setup example for two external IRQ

```
void irq_task_code(void)
{
    int i=0;
    printf("IRQ Task starts\n ");

    while(1)
    {
        sierra_await_irq(1); //Wait for external IRQ 1
        printf("IRQ 1 start\n");
        for(i=0; i<500000; i++); //virtual load
        sierra_await_irq(0); //Wait for external IRQ 0
        printf("IRQ 0 start\n");
        for(i=0; i<500000; i++); //virtual load
    }
}
```

Time Management

This section describes the functionality of the time management controller. The following functions are implemented:

sierra_delay_task
sierra_period_time_init

sierra_await_next_period

Description

Blocks the calling task specified number of ticks. The task will be placed in the blocked state until the timer expires or an undelay call is performed on the task.

When the timer expires, or if the undelay call is performed, the task is placed in the ready state.

Function declaration

```
void sierra_delay_task (int delay_time)
```

Argument

delay_time	Specifies the number of ticks to delay the task. Max value depends on the version of Sierra.
------------	---

Return codes

N/A

Example

```
void t1(void)
{
    while(1)
    {
        sierra_delay_task(10); /* t1 is blocked
                               for 10 ticks */
    }
}
```

sierra_period_time_init

Description

Initialize the period time for the calling task. This function must be performed before the use of the function *sierra_await_next_period()*. See the version of Sierra for the max value. Possible to use deadline control, to detect starvation.

Function declaration

```
void sierra_period_time_init (int period_time)
```

Argument

Period_time Specifies the period time, in number of ticks, for calling task.

Return codes

deadline_control

N/A

Example

```
void t1(void)
{
    sierra_period_time_init(100); /* Initialize period
                                   time for t1 to
                                   100 ticks */

    while(1)
    {
        /* Insert code here */
    }
}
```

sierra_await_next_period

Description

Suspends a periodic task until the start of next period time. If you miss a periodic start, Sierra will skip this period, not to disturb the other tasks, the miss to the periodic task will be reported. The deadline is the same as the period time.

To use deadline control cost no extra execution or response time to manage.

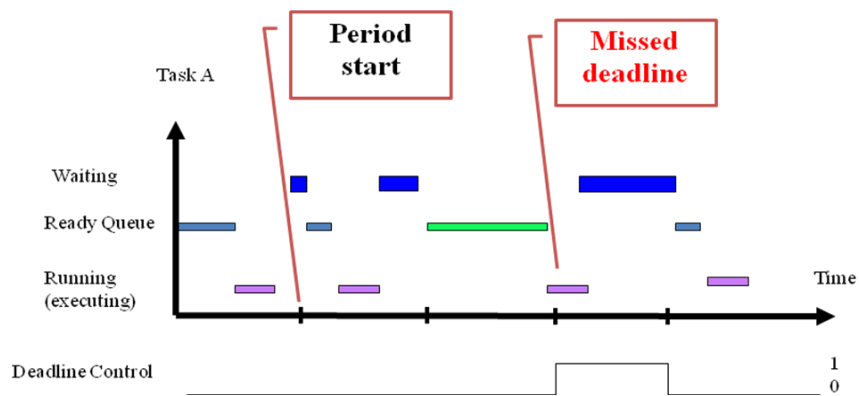


Figure 5. Periodic start with deadline control.

Function declaration

```
void sierra_await_next_period (void)
```

```
task_periodic_start_union sierra_await_next_period  
(void)
```

Argument

deadline_control:

0: Ok, deadline not missed.

1: missed at least one deadline.

Return codes

N/A

Example

```
// Without deadline control  
void t1(void)  
{  
    init_period_time(50);  
  
    while(1)  
    {  
        sierra_await_next_period();  
    }  
}  
  
// With deadline control  
task_periodic_start_union test;  
  
while(1)  
{  
    test = sierra_await_next_period();  
  
    if (test.periodic_start_integer & 0x1)  
        printf("deadline miss, timer task");  
}
```

Semaphore Management

This section describes the functionality of the semaphore management. The semaphores are used in the system to protect shared resources and for synchronization of different tasks.

There are 8 binary semaphores available in the Sierra. A semaphore can have a queue of waiting tasks of the same length as the number of tasks in the system. This means that a semaphore can be taken by one task and up to 8 other tasks can be waiting for it. The queue is arranged by task ID numbers. Task with highest ID number in the queue will run when the semaphore becomes available.

The following semaphore handling functions are supported:

```
sierra_take_sem  
sierra_release_sem  
sierra_read_sem
```

sierra_take_sem

Description

Makes a task pending (waiting) for a semaphore. If the semaphore is free, the task will continue to execute immediately. If the semaphore is allocated by another task, the calling task will be suspended and put in a semaphore waiting queue, until the semaphore becomes free.

Note: The queue is arranged in task ID numbers and task with highest ID number in the queue will get the semaphore when it becomes available.

Function declaration

```
void sierra_take_sem (int semID)
```

Argument

semID	Semaphore number (0-15)
-------	-------------------------

Return codes

N/A

Example

```
#define SEM1 1  
  
void t1(void)  
{  
  
    while(1)  
    {  
        sem_take(SEM1); /* Pend on semaphore 1 */  
    }  
}
```

sierra_release_sem

Description

Releases a specified semaphore. If there are one or more tasks waiting for the semaphore, the first task in the semaphore waiting queue will get the semaphore and will be moved to ready state.

Function declaration

```
void sierra_release_sem (int semID)
```

Argument

semID	Semaphore number
-------	------------------

Return codes

N/A

Example

```
#define SEM1 1

void t1(void)
{
    while(1)
    {
        sem_release(SEM1); /* Release semaphore 1 */
    }
}
```


sierra_read_sem

Description

Read a task's semaphore status.

Function declaration

```
Sem_info_t sierra_read_sem (int taskID)
```

Argument

taskID	Specifies the taskID to read status of.
--------	---

Return codes

Sem_info_t

status	0 = The task is not waiting for a semaphore (ignore semID) 1 = The task is waiting for a semaphore (Read semID)
semID	Semaphore number if specified task is waiting for a semaphore.

Example

```
#define SEM3 3

void t1(void)
{
    sem_info_t sem;
    int semID, status;

    while(1)
    {
        sem = sem_read(T2); /* Read semaphore status of
                             task T2 */

        /* The different member variables in the
           returned data-structure: */
        status = sem.status;
        semID  = sem.semID;
    }
}
```

Flag Management

The Sierra has support for flags for efficient synchronizing of events. The entire synchronizing algorithm is handled by the hardware kernel. This makes handling of flags very efficient since no valuable CPU time is spent on synchronization.

Flags are very efficient in cases where you, for example, have one or several events handled by some input tasks and there exist an output task triggered by one or several tasks - see figure 8 below.

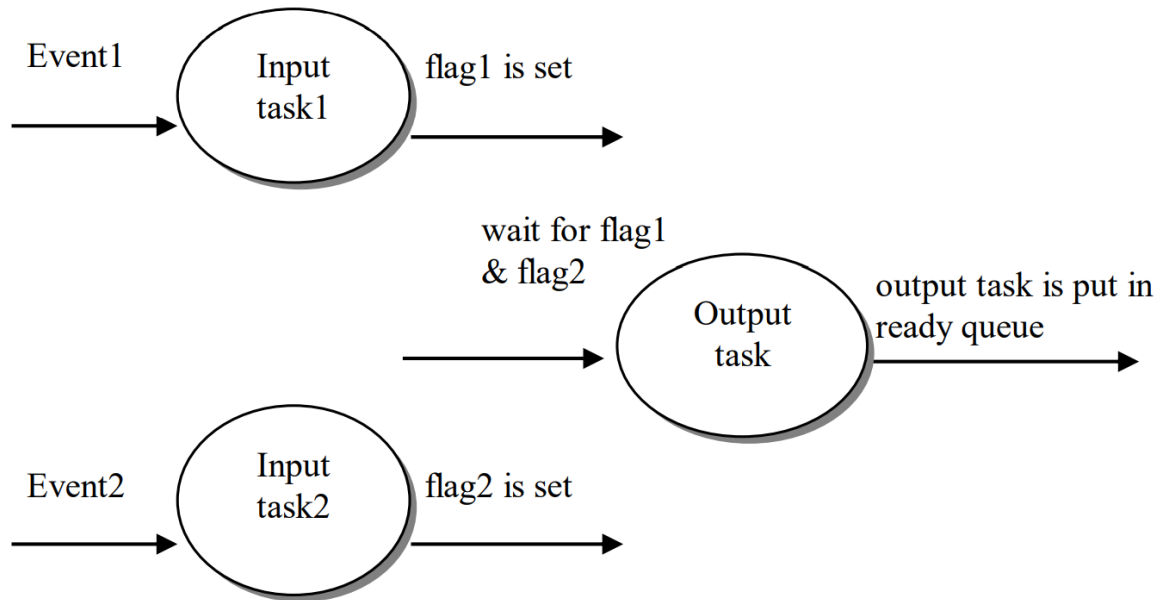


Figure 6. Flag example.

The semantics for the figure is; the output task makes a system call where it will need a combination of flags set to be able to continue to run. If this combination is not true at the time when the call is performed, the task will be suspended until the combination becomes true. Task1 runs and sets flag1. At this point the output task will not be made ready, as it asks for an *AND* operation between flag1 and flag2. When task2 has set flag2, the output task will be made ready. The output task is scheduled and will start to run when it has the highest priority in the ready queue.

If the Sierra is configured to support 4 flag bits, the flag bits can be used in 2^4-1 (=15) different combinations.

The following flag handling functions are supported:

- sierra_await_flag
- sierra_set_flag
- sierra_clear_flag

sierra_await_flag

Description

This call makes a task wait for one or more flags to be set. If the flag(s) are already set, the task will continue to run, if not it will be suspended until the combination is set.

Function declaration

```
void sierra_await_flag (int flag_mask)
```

Argument

flag_mask

The four lowest bits are used, values between 1-15 are valid. 0 is not a valid flag value.

Return codes

N/A

Example

```
#define FLAG_MASK 5 /* Flag1 AND Flag3 -> 0101 */

void t1(void)
{
    while(1)
    {
        sierra_await_flag(FLAG_MASK); /* Wait for Flag1
                                       and Flag3 to be
                                       set */
    }
}
```

sierra_set_flag

Description

This call sets one or more flags. If there are any task(s) waiting for the specific combination of flags that are set during the call, they will be made ready and start to run when they have the highest priority in the ready queue.

If a task is waiting for a combination of flags and the call only sets one or few of the flags, the waiting task will not be activated before all flags are set.

Function declaration

```
void sierra_set_flag (int flag_mask)
```

Argument

flag_mask

The four lowest bits are used, values between 1-15 are valid. 0 is not a valid flag value.

Return codes

N/A

Example

```
#define FLAG_MASK 7          /* Flag1 AND Flag2 AND
                             Flag3 -> 0111 */

void t1(void)
{
    while(1)
    {
        sierra_set_flag(FLAG_MASK); /* Set Flag1, Flag2
                                     and Flag3 */
    }
}
```

sierra_clear_flag

Description

This call clears one or more flags. When a flag has been set, it needs to be cleared after a waiting task has taken care of the event that was waiting for the flag. If there is more than one task using the flag, it is important to know which one(s) of these tasks that will be permitted to do this call.

Example; there are two tasks waiting for a common flag, but one of the tasks is also waiting for another flag. When this flag is set, the task that only waits for *this* flag is made ready and will start to run when it has the highest priority in the ready queue. However, if the other task still is waiting for the other flag when this first task has done its job, this first task should not clear the flag as the other task still is depending on this flag. In this specific scenario it is the task that is waiting for both flags that should clear the flag.

Function declaration

```
void sierra_clear_flag (int flag_mask)
```

Argument

flag_mask The four lowest bits are used, values between 1-15 are valid. 0 is not a valid flag value.

Return codes

N/A

Example

```
#define FLAG_MASK 7          /* Flag1 AND Flag2 AND
                             Flag3 -> 0111 */

void t1(void)
{
    while(1)
    {
        sierra_clear_flag(FLAG_MASK); /* Clear Flag1,
                                     Flag2 and
                                     Flag3 */
    }
}
```

Sierra information calls

Sierra HW Version

Description

Sierra Version number can be retrieved from Sierra Hardware if you call `sierra_HW_version` function.

- MAJOR version when you make incompatible changes,
- MINOR version when you add functionality in a backwards-compatible manner
- PATCH version when you make backwards-compatible bug fixes
- Number of tasks

Table 1: Sierra version register (binary)

31-28	27-24	23 - 20	19 - 16	15-8	7 - 0
MAJOR_version	MINOR_version	PATCH_version	X	Number of semaphores	Number of tasks

Function declaration

```
version_register_union sierra_HW_version(void)
```

Argument

N/A

Return codes

```
version_register_union    number_of_tasks (8 bits)
                           number_of_semaphores (8 bits)
                           N\A (4 bits)
                           PATCH_version (4 bits)
                           MINOR_version (4 bits)
                           MAJOR_version (4 bits)
```

Example

```
#include "altera_avalon_sierra_ker.h"

void sierra_hw_version_print(void)
{
    version_register_union test = sierra_HW_version();

    printf("Sierra HW Major Version = %d\n",
test.version_register.MAJOR_version);
void sierra_hw_versions(void)
}
```

Return:

Sierra HW Major Version = 9

Example

```
Void Printf_sierra_HW_version(void)
```

Return:

Version = 9.5.0
Number of task bits = 3 - 8 tasks

Number of semaphore's bits = 3 - 8 semaphores

Sierra SW Version

Description

Sierra Version number from Sierra software can be retrieved if you call `sierra_SW_driver_version()` function.

Function declaration

```
sw_version_union sierra_SW_driver_version(void)
```

Argument

N/A

Return codes

sw_version_union	PATCH_version (10 bits)
	MINOR_version (10 bits)
	MAJOR_version (12 bits)

Example

```
#include "altera_avalon_sierra_ker.h"

void sierra_sw_version_print(void)
{
    sw_version_union info = sierra_SW_driver_version();

    printf("  Sierra SW  Major Version = %d\n",
test.version_register.MAJOR_version);
}
```

Return:

Sierra SW Major Version = 10.03.15

Print Sierra Versions

Description

To print Sierra version numbers for hardware and software you can call `sierra_print_versions()` function. The version numbers are divided into three sections.

- MAJOR version when you make incompatible changes,
- MINOR version when you add functionality in a backwards-compatible manner
- PATCH version when you make backwards-compatible bug fixes

Function declaration

```
void sierra_print_versions(void)
```

Argument

N/A

Return codes

N/A

Example

```
#include "altera_avalon_sierra_ker.h"

void sierra_versions_print(void)
{
    sierra_print_versions();
}
```

Return:

```
Sierra HW version 9.4.2
Sierra SW version 10.3.15
```

Logging API

Logging macros for three verbosity level. The levels implemented are `SIERRA_LOG_INFO`, `SIERRA_LOG_WARN` and `SIERRA_LOG_ERROR`.

The system would save messages from different Sierra functions and then print the results using a logging subsystem. The info level would log events during normal execution of a program and provide the user with descriptive data of how the system operated. The warning level would log incidents and abnormal events prior to execution and during runtime. The error level would address more serious concerns, for example, if the system encounters a problem from which it cannot recover.

Sierra Time Logging Register

The time register was built as a part of the Sierra hardware, and that count Sierra ticks directly from the Sierra kernel, without any external interference. The register is defined as a vector (31 DOWNT0 0), and it counts the internal time ticks (0 - 268 435 454 (dec)). After it has come to the maximum it starts from 0 again.

The function `sierra_get_current_time(void)` return the value from the time logging register.

Table 2: An overview of the time logging register

Description	Utilization
<code>#define M_RD_TIME_LOGGING_REGISTER</code> <code>IORD_32DIRECT(SIERRA_RTOS_BASE, 0x70);</code>	Macro for reading data from hardware logging register
<code>uint32_t</code> <code>sierra_get_current_time(void)</code>	Function for returning a value from the register

ON/OFF Logging System

`SIERRA_LOGGING`, was created to regulate the system from being on or off. In `sierra_logging.h` the user can turn on and off the logging system by updating its defined value:

0 – logging is disabled

1 – logging is enabled

2 – logging is enabled with timestamps

Logging functions

The logging subsystem incorporates four different logging functions, one for each verbosity level and one for notifying the user about the current status of the logging interface. The functions print the results (info, warning or error) from Sierra functions using variadic macros.

Table 3: Logging functions from subsystem

Declaration	Usage
void sierra_print_logging_status (void)	Prints current logging status. If the macro <code>SIERRA_LOGGING</code> is defined then a message will be printed saying logging is active. If not defined, then the message will say the logging interface is not active.
void sierra_log_info (const char* szMsg)	The function prints informative messages and presents data from Sierra functions during normal execution of a library. If conditions are met, the function will also print the time.
void sierra_log_warn (const char* szMsg)	The function prints warning messages and presents metadata from Sierra functions that encounter abnormal activity. If conditions are met, the function will also print the time.
void sierra_log_error (const char* szMsg)	The function prints error messages and presents metadata from functions that have encountered an erroneous event. If conditions are met, the function will also print the time.

Sierra Logging Probes

Logging probes were implemented to retrieve data from Sierra functions, and these would be triggered if certain conditions were met during execution. The probes would have messages written to the logging macros before being sent to the subsystem. Next figure shows an example of logging data from Sierra.

```

LOG_INFO: SIERRA_TASK, Next task requested, task 0 moved to Running state TIME: 130
.LOG_INFO: SIERRA_TASK, task 0 replaced task 2 TIME: 152
LOG_INFO: SIERRA_TASK, task 0 replaced task 4 TIME: 152
LOG_INFO: SIERRA_TIME, Task 2 is suspended until next time period TIME: 156
LOG_INFO: SIERRA_TASK, Next task requested, task 2 moved to Running state TIME: 156
LOG_INFO: SIERRA_SVC, sem_take 1 TIME: 156
LOG_INFO: SIERRA_SVC, sem_release 1 TIME: 157
LOG_INFO: SIERRA_TIME, Task 0 is suspended until next time period TIME: 157
LOG_INFO: SIERRA_TASK, Next task requested, task 0 moved to Running state TIME: 157
..LOG_INFO: SIERRA_TASK, task 0 replaced task 4 TIME: 177
LOG_INFO: SIERRA_TIME, Task 0 is suspended until next time period TIME: 177
LOG_INFO: SIERRA_TASK, Next task requested, task 0 moved to Running state TIME: 177
..LOG_INFO: SIERRA_TASK, task 0 replaced task 2 TIME: 202
LOG_INFO: SIERRA_TASK, task 0 replaced task 4 TIME: 202
LOG_INFO: SIERRA_TIME, Task 2 is suspended until next time period TIME: 203
LOG_INFO: SIERRA_TASK, Next task requested, task 2 moved to Running state TIME: 204
LOG_INFO: SIERRA_SVC, sem_take 1 TIME: 204
LOG_INFO: SIERRA_SVC, sem_release 1 TIME: 204
LOG_INFO: SIERRA_TIME, Task 0 is suspended until next time period TIME: 204
LOG_INFO: SIERRA_TASK, Next task requested, task 0 moved to Running state TIME: 204
..LOG_INFO: SIERRA_TASK, task 0 replaced task 4 TIME: 227
.LOG_INFO: SIERRA_TASK, task 0 replaced task 2 TIME: 227
LOG_INFO: SIERRA_TIME, Task 0 is suspended until next time period TIME: 227
LOG_INFO: SIERRA_TASK, Next task requested, task 0 moved to Running state TIME: 230

```

Figure 7; Example of logging data from Sierra (more information;
https://www.youtube.com/watch?v=zb9rq_gIQNI)

Extended API

Mailbox

This section describes the functionality of the Mailbox. The Mailbox is used for sharing data between tasks/processes in safe manner that won't cause deadlocks or race conditions.

Mailbox is flexible to fit user needs but there are some limitations, such as 65 536 message per Mailbox with size limit of 65 536 bytes per message. Following Mailbox is implemented:

- `sierra_mbox_get_required_size`
- `sierra_mbox_init`
- `sierra_mbox_send`
- `sierra_mbox_read`
- `sierra_mbox_peak`

`sierra_mbox_get_required_size`

Description

Calculate size for char array that will be used for Mailbox.

Function declaration

```
extern uint32_t sierra_mbox_get_required_size(const uint16_t amount_of_messages, const uint16_t largest_message_data)
```

Argument

`max_messages`, Maximum messages.

`largest_size`, Largest message data.

Returns

`uint32_t`, that represents required size for mailbox based on inputs.

Notes/Warnings

Nothing.

sierra_mbox_init

Description

Initiates Mailbox to initial state based on passed values.

Function declaration

```
extern void sierra_mbox_init(sierra_mbox_queue_t* mbox, char* mem_pool, uint16_t max_messages, uint16_t largest_size)
```

Argument

mbox Mailbox to initialize.

mem_pool char array that will be used to store header and message/data.

max_messages how many messages can mem_pool hold.

largest_size how large will message/data be.

Returns

Nothing

Notes/Warnings

Make sure that mem_pool (aka char array) will be able to hold all the messages, the example code will show how to get accurate size for mem_pool.

Example

```
#include <sierra_extension/sierra_mbox.h>
...
// create global Mailbox that is accessible for tasks.
sierra_mbox_queue_t mbox;
...
void main()
{
    // calculate total size needed for memory pool
    uint16_t amount_of_message = 3; // amount of message that Mailbox can save.
    uint16_t largest_message_data = 4; // the maximum size of the message can be saved.

    // create memory pool for Mailbox
    char memmory_pool[sierra_mbox_get_required_size(amount_of_message, largest_message_data)];
    sierra_mbox_init(mbox, memmory_pool, amount_of_message, largest_message_data);
}
```

sierra_mbox_send

Description

Copies header content byte by byte to Mailbox, same with headers data that it is pointing to.

Function declaration

```
extern sierra_mbox_res_e sierra_mbox_send(sierra_mbox_queue_t* mbox, const sierra_mbox_header_t* header)
```

Argument

mbox Which Mailbox to send.

header What to save.

Returns

Returns one of the possible responses from sierra_mbox_res_e, if everything went well response will be MBOX_OK otherwise different response.

Notes/Warnings

Headers data can point to anything, it can point to struct and it will be saved to Mailbox if there is enough space in Mailbox.

Example

```
#include <sierra_extension/sierra_mbox.h>
...
// create global Mailbox that is accessible for tasks.
sierra_mbox_queue_t mbox;
...
void task_send()
{
    ...
    sierra_mbox_header_t header;
    header.id = 1;
    ...
    char msg[18] = "My lucky number is";
    int lucky_num = 7;
    ...
    while(1)
    {
        ...
        header.type = MBOX_CHAR_ARR;
        header.size = sizeof(msg);
        sierra_mbox_send(&mbox, &header);
        ...
        header.type = MBOX_INTEGER;
        header.size = sizeof(lucky_num);
        sierra_mbox_send (&mbox, &header);
    }
}
```

sierra_mbox_read and sierra_mbox_peak

Description

Copies stored header and data that is in Mailbox to users/passed header and headers data.

Read removes the header and headers data after reading, Peak just reads the headers and headers data without removing from Mailbox.

Function declaration

extern sierra_mbox_res_e **sierra_mbox_read**(sierra_mbox_queue_t* mbox, const sierra_mbox_header_t* header)
or extern sierra_mbox_res_e **sierra_mbox_peak**(sierra_mbox_queue_t* mbox, const sierra_mbox_header_t* header)

Argument

mbox Which Mailbox to read or peak.

header where to save header and headers data.

Returns

Returns one of the possible responses from sierra_mbox_res_e, if everything went well response will be MBOX_OK otherwise different response.

Notes/Warnings

Headers data needs to point user defined char array with enough space for message/data.

Example

```
#include <sierra_extension/sierra_mbox.h>
...
// create global Mailbox that is accessible for tasks.
sierra_mbox_queue_t mbox;
...
void task_read()
{
    ...
    sierra_mbox_res_e res;
```

```

...
char buffer[50];
sierra_mbox_header_t header;
header.data = buffer;
...
while(1)
{
    ...
    res = sierra_mbox_read(&mbox, &header);
    if (res != MBOX_OK)
    {
        printf("something went wrong\n");
        continue;
    }
    ...

    if (header.type == MBOX_INTEGER)
        printf("%d\n", *(int*)header.data);

    if (header.type == MBOX_CHAR_ARR)
    {
        for(int i = 0; i < header.size; i++)
            printf("%c", *((char*)header.data + i));
    }
}
}

```

Hardware interface

Sierra is a component with bus interface, running task ID information, external interrupt and external start of blocked tasks.

Bus interface (TDBI) can be wrapped to the most busses on the market.

Running task ID info can be used to monitor the running task or logged of another hardware units for different types of analyses.

External interrupt is direct connected to a task and the task will be scheduled before it can be running on the CPU

External start of blocked task is an advanced function to communicate with hardware units connected to SW tasks.

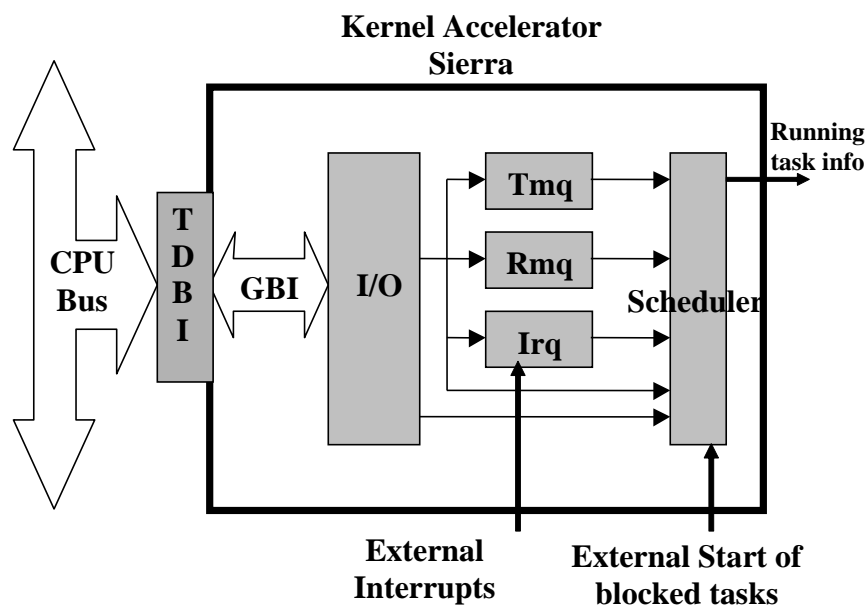


Figure 8. Block schematic.

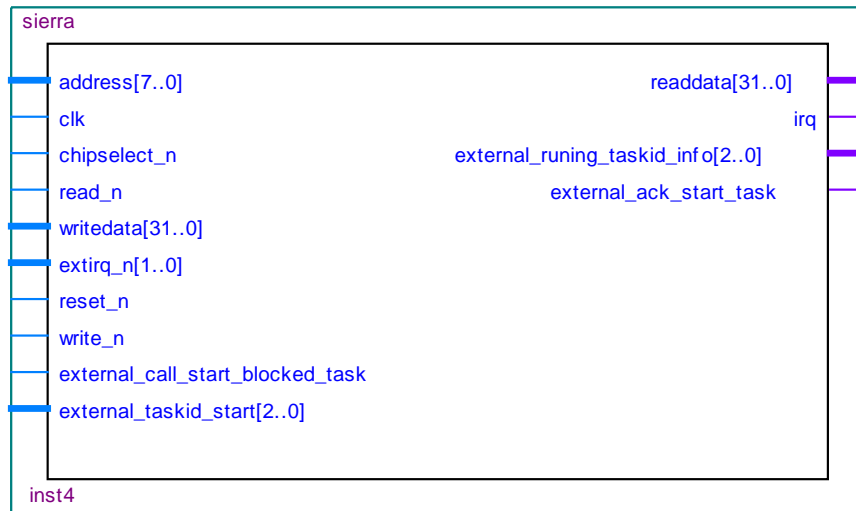


Figure 9: Sierra pin Interfaces

Table 4 Sierra Pin out

Pin name	Direction	Description
clk	Input, Sys	System clock
reset_n	Input, Sys	HW reset
cs_n	Input, Bus	Chip select
write_n	Input, Bus	Read / Write
addr(7:0)	Input, Bus	Address bus
din(31:0)	Input, Bus	Data bus in
dout(31:0)	Output, Bus	Data bus out
irq_n	Output, CPU	Task switch interrupt
extirq_n(1:0)	Input, User	External interrupts
External_runing_taskid_info[2..0]	Output, User	Updating Running task ID (binary)
external_call_start_blocked_task (extended Sierra)	Input, User	Start of Blocked Tasks, Not used = '0'.
external_ack_start_task (extended Sierra)	Output, User	Start of Blocked Tasks
external_taskid_start[2..0] (extended Sierra)	Input, User	Start of Blocked Tasks

Protocol with external start of blocked task (extended Sierra)

Start of blocked task is done with following protocol:

- 1) Set "external_taskid_start" that should be started (it have to be in block state, block_task()) and write '1' to "external_call_start_blocked_task"
- 2) Wait for "external_ack_task_start" to be '1'
- 3) Write '0' to "external_call_start_blocked_task"
- 4) Wait for "external_ack_task_start" to be '0'

Sierra SW File Structure

The Sierra low level API SW consists of the following files:

- ❏ Sierra_RTOS (inte uppdaterad)
 - ❏ HAL
 - ❏ inc
 - altera_avalon_sierra_io.h
 - altera_avalon_sierra_ker.h
 - altera_avalon_sierra_name.h
 - altera_avalon_sierra_regs.h
 - altera_avalon_sierra_tcb.h
 - altera_avalon_sierra_tcb_offset.h
 - sierra_logging.h
 - ❏ src
 - csw.S (context swtich routine)
 - sierra.c (basic service calls)
 - sierra_sem.c (Semafres and flags service calls)
 - sierra_taskc (Task service calls)
 - sierra_time.c (Time service calls)
 - sierra_info.c (Extra logging/info service calls)
 - sierra_logging (logging functions)
 - alt_exception_enty.S (NiosII exception handling)
 - alt_exception_trap.S (NiosII trap handling)
 - component.mk (makefile)