User Reference Manual, ver 9.4.1

Sierra

This manual covers the use of Sierra RTK and al function perhaps is not 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

About This Manual

The Sierra RTK consists of two parts:

1. Sierra IP (Intellectual Property) HW
2. Sierra API (Drivers) SW

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.

Purpose

The purpose of this Users Reference Manual is to give system designers using the Sierra Real-Time Operating System an overview and functional description of how it works. Some examples are included in this document as a help to getting started.

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, it's state, stack owned resources, the value of the processor registers etc.

Sierra RTK - General Description

Configuration of the API

This chapter gives short overview of the Sierra RTOS functionality. The Sierra HW core is partitioned into modules as shown in the figure and described in the text below.

Tmq = Time Manager
Rmq = Resource Manager
Irq = Interrupt Handler
TDBI = Technology Dependent Bus Interface
GBI = General Bus Interface

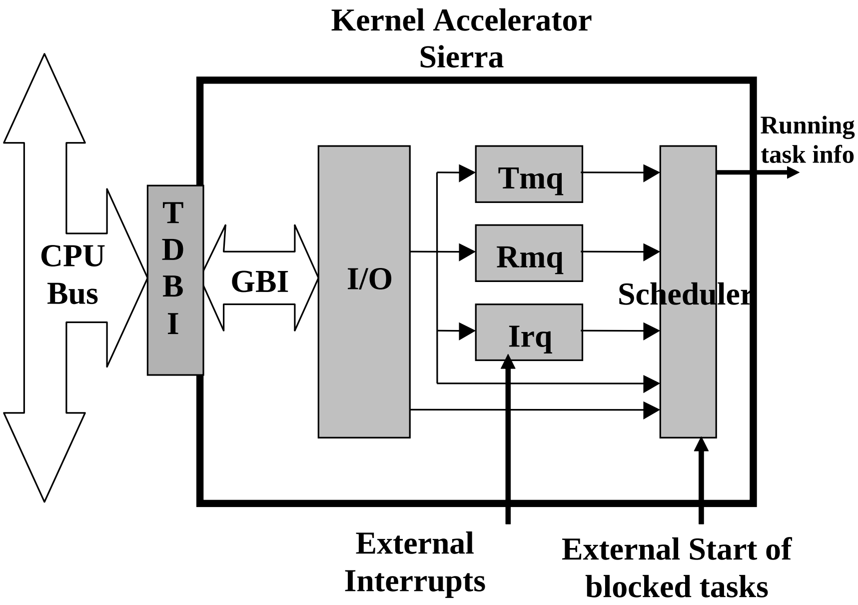

Figure 1 Overview of internal blocks in Sierra kernel.

Core Engine

Sierra RTOS is partitioned into these functional units:

Interface
Scheduler
Semaphore and Flag Handler
Time Management Controller

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.

Sierra Interface

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.

It also pin out that reflect the running task id. Those can be used to drive led diodes, statistics etc.

In Sierra also interrupt controller is included, that can trig direct a task without any software.

Also external hardware start of blocked task. This can be used in advanced system with hardware drivers.

Sierra Configuration

The Sierra is a small complete RTOS kernel with support for task handling, semaphores, timers and external interrupts. All operations are carried out in the Sierra chip, and the software that comes with the package is a driver for communication between the CPU and hardware kernel.

The Sierra handles:

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

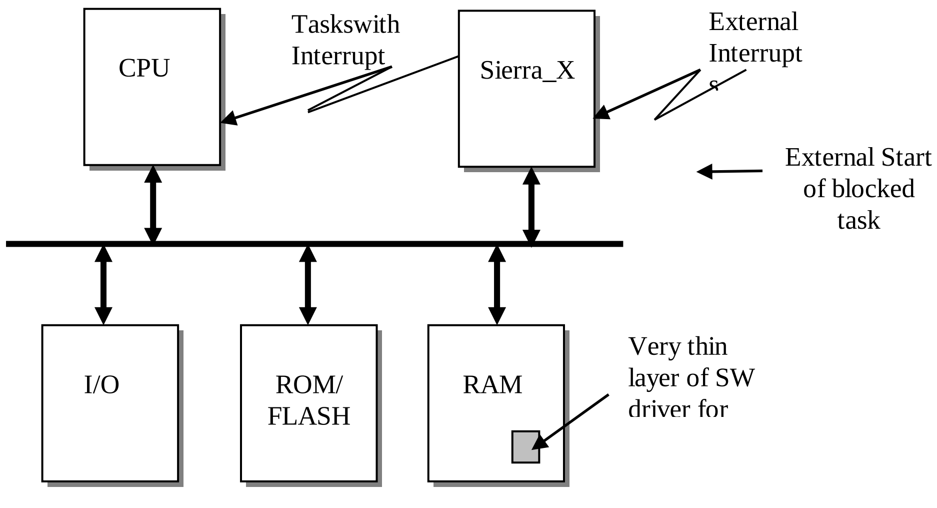

Figure 2 Example of system configuration.

Scheduler

The Scheduler unit controls all scheduling in the Sierra. The scheduler can handle tasks at different priority levels. See the configuration table.

Normally a number of tasks are created when the system is initialized. Among these tasks an idle-task must be created. This task will execute when there is no other task ready to execute. If no idle-task would exist the system behavior would be undefined in the case that no task is ready to execute. The idle-task shall not make any system calls to the Sierra.

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). Tasks must have a priority and the priority must be initialized when the task is created. When a task is created it must be given a unique task-ID so the Sierra can separate the tasks.

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. When a task is running, there are some events that can change the tasks state to another state, see below.

1. The task asks for a task-switch itself

2. The task tries to lock a semaphore that is already locked and the task becomes blocked.

3. A task with higher priority is becoming ready and therefore pre-empts the task and thereby placing the running task back into the ready-queue.

The Sierra RTOS supports the following task management functions:

enable/disable task switch
create a task
start task
delete task
block/start task
get task information

The task management also support running task id number connected to hardware pins. For more information see the datasheet.

Task states

The Sierra can support the following task states and transitions:

Running
Ready
Blocked /Waiting
Wait for interrupt
Dormant

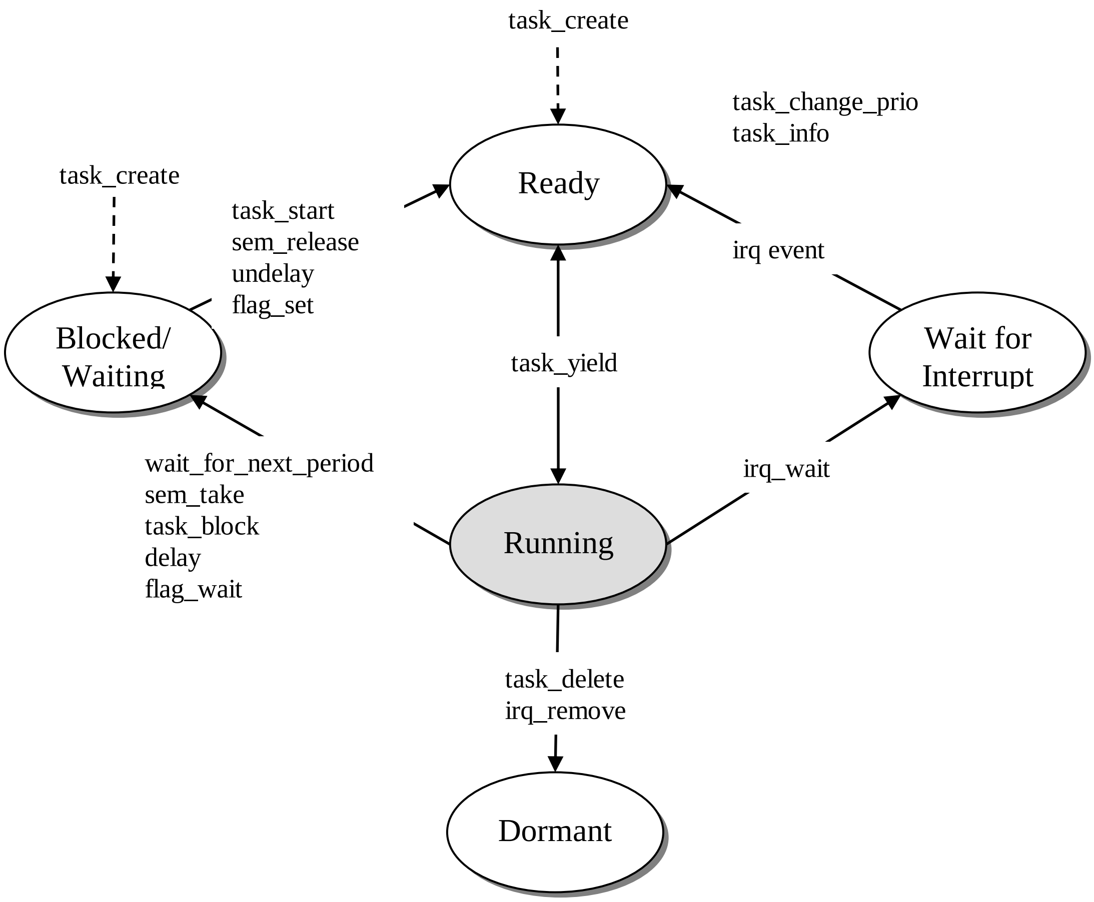

Figure 3: State transitions in Sierra

Time Management Controller

In the Sierra, the following time-handling functions are implemented, delay, undelay and support for periodic tasks. In a system where these functions are implemented in software they will increase the system-overhead.

In ordinary software RTOS the principle for the delay functionality is as follow. Every clock-tick the RTOS has to check the timer-queue and decrease each tasks timer and examine if any timer has expired. When a timer expires for a task it has to be scheduled in to the ready-queue. All this calculation has a cost in time and this time increases with number of tasks using the timers.

When it comes to the Sierra all this handling with timers etc. is done inside the Sierra and all cost in time have been removed from the system.

The Sierra supports the following time management functions:

set timebase register
initialize periodic time
wait for next period
delay

Semaphore and Flag Handler

The Sierra supports the use of 8 binary semaphores and 8 flags as synchronizing functions. Semaphores are used to synchronize resources in a system that is shared between tasks. A resource can be an I/O port, a memory area etc.

Flags are used to synchronize events between tasks. Flags are very useful as many tasks can be triggered by one flag at the same time. For example; More than one task is waiting for a result that a certain task produces. By setting a flag, the producer activates all waiting tasks at the same time and the scheduler decides which of them should run according to priority, place in queue etc.

These semaphore functions are supported:

sem_take
sem_release
sem_read

These flag handling functions are supported:

flag_wait
flag_set
flag_clear

Information, Setup and Initiating

Sierra Hardware/Software Initiation

Description

Initiate the TCB in soft/hardware and also reset the Sierra hardware. This can always use to make a reset of SW drivers and HW based sierra. All tasks, etc. kills and also the task switch is disable, also the TCB will be cleared. After instantiation the task switch is off. This is done in less than 100 system clock for the standard Sierra IP.

Function declaration

```
void Sierra_Initiation_HW_and_SW(void)
```

Argument

Nothing

Return codes

Nothing

Sierra HW Version

Description

Sierra Version number can you get 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

```
unsigned int sierra_HW_version(void)
```

Argument

Nothing

Return codes

Unsigned int

Example

```
printf("Sierra HW version = %d\n",
sierra_HW_version());

Return:
Sierra HW version = -1827667965 (unsigned int)
```

Special print function (info.c)

```
Void Printf_sierra_HW_version(void)

Return:
Version = 9.3.1
Number of task bits = 3
Number of semaphore's bits = 3
```

Sierra SW Version

Description

Sierra Version number can you get from Sierra Hardware if you call Sierra_SW_Driver_version() function.

Function declaration

```
unsigned int sierra_SW_driver_version(void)
```

Argument

Nothing

Return codes

Unsigned int

Example

```
printf("  Sierra SW version = %d\n",
sierra_SW_driver_version());
```

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.

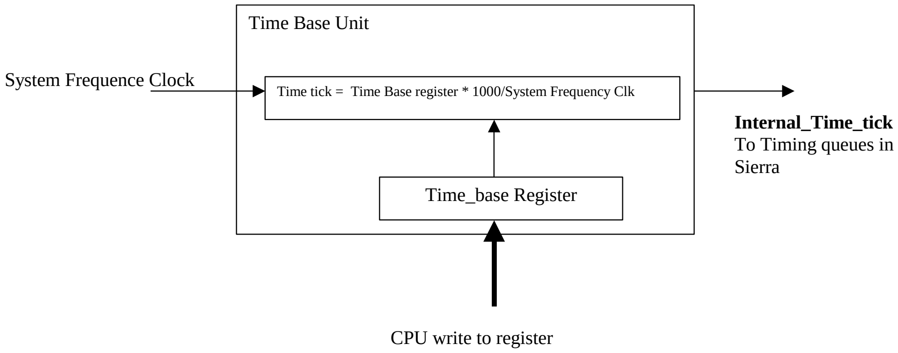

Figure 4 Time Base Unit

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

Function declaration

```
//Set time_based register.
void set_timebase (unsigned int time_set)

//Read Time_base register
unsigned int SierraTime_base_reg(void)
```

Argument

```
//Set time_based register, see sierra specification
for number of bits:
//13 bits; time_set: range 0-8191, please check the
version of hardware.
unsigned int

//Read Time_base register
Nothing
```

Return codes

```
//Set time_based register (32 bits)
Nothing

//Read Time_base register
unsigned int
```

Example

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

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 (task_create)
Starting of tasks (task_block)
Yield (task_yield)
Get task status (task_getinfo)
Task switch off and on (tsw_on and 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 has to be created with task id 0 and lowest priority (0).

task_create

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 has to be created and have task id 0 and lowest priority (0).

Function declaration

```
void task_create (      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 taskID=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

Nothing

Notes/Warnings

Nothing

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)
{
   :
task_create(T1, PRIO1, READY, t1, stack1, STACK1_SZ);
   :
}
```

task_start

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 task_start (int taskId)
```

Argument

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

Return codes

Nothing

Example

```
:
#define T2 2
:

void t1(void)
{

   task_start(T2); /* t1 starts T2 */

   while(1)
   {
     :
     :
   }
}
```

task_getinfo

Description

Get status information about a specified task.

Function declaration

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

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

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 tsw_off (void)`

Argument

N/A

Return codes

N/A

Example

```
void t1(void)
{

   while(1)
   {
      :
      tsw_off();     /* Entering critical section -
                        Turn off task-switch interrupts
                      */
      :
      :
   }

}
```

tsw_on

Description

Enables task-switch interrupt.

Function declaration

```
void 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 */
    :
  }
}
```

task_block

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 task_block (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){task_block(T2);i = 0;}
 /* Block t2 when i==10 */
}
}
 }
```

task_change_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 **task_change_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. “1” is the lowest-priority level for user tasks (IDEAL has “0”).

Return codes

N/A

example

```
:
#define T2 2
:
#prio_5 5
:

void t1(void)
{

   task_change_prio(T2,prio_5); /* Task T2 get
priority 5 */

   while(1)
   {
     :
     :
   }
}
```

task_delete

Description

Delete the currently running task. The task will be moved from the system and the tasl id number will be free to be used again. Most be created again to start. It is **not allowed** to perform this call from the idle task.

Function declaration

```
void task_delete(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.

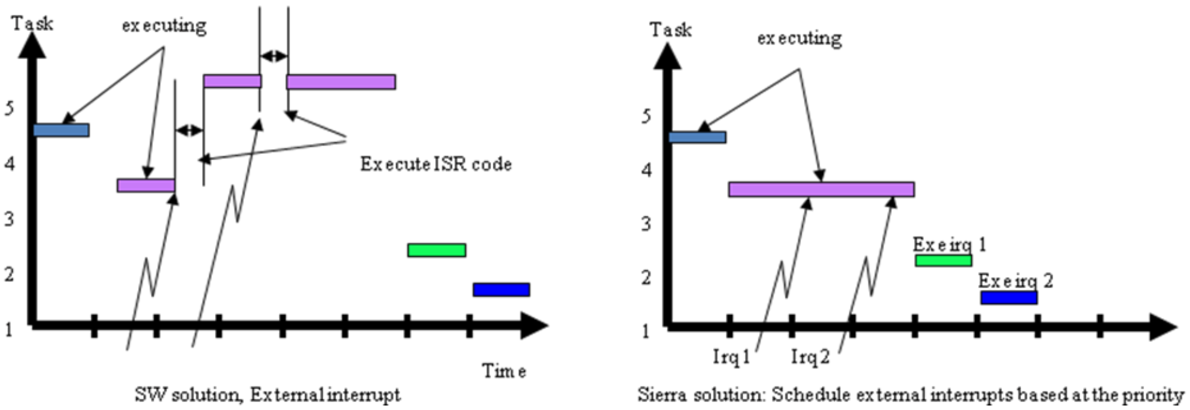

Figure 5: SW RTK and HW based Sierra solution, two low priority irq.

Wait for interrupt

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 irq_wait(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

Nothing

Example

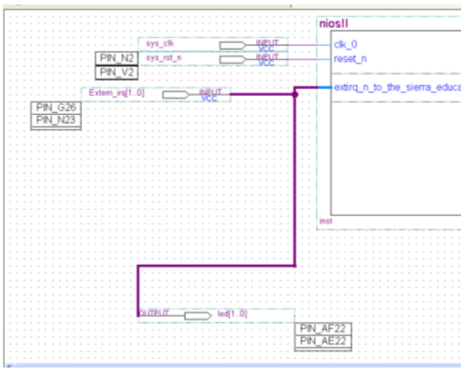

Figure 6: Example of two external irq

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

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

delay
init_period_time
wait_for_next_period delay

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 delay (int delay_time)
```

Argument

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

Return codes

Nothing

Example

```
void t1(void)
{
    :

    while(1)
    {

        :
        delay(10); /* t1 is blocked for 10 ticks */
        :
    }

}
```

init_period_time

Description

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

Function declaration

```
void init_period_time (int period_time)
```

Argument

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

Return codes

Deadline_control

Nothing

Example

```
void t1(void)
{
    :

    init_period_time(100); /* Initialize period time
                              for t1 to 100 ticks */

    while(1)
    {

        :
        :

    }
}
```

wait_for_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 disturbed the other tasks and also report the miss to the periodic task. The deadline is the same as the period time.

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

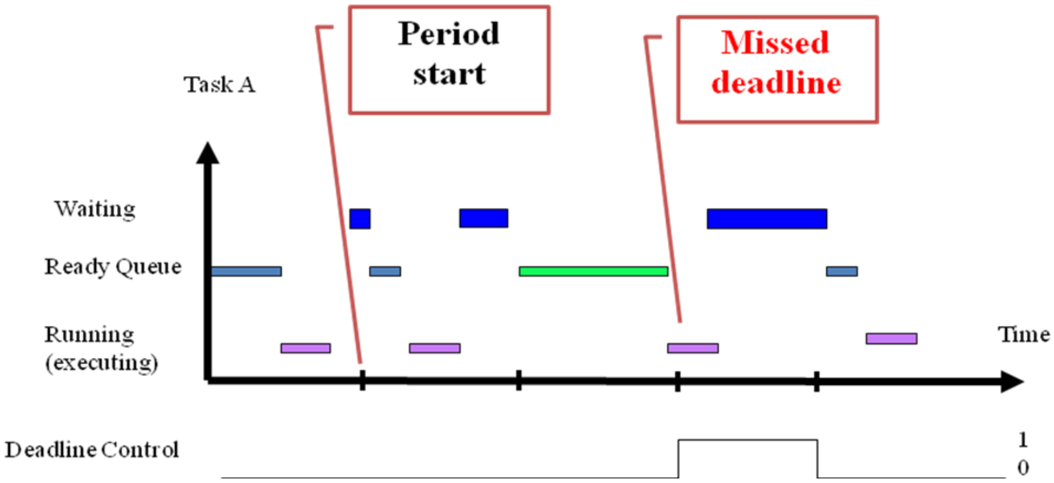

Figure 7: Periodic start with deadline control

Function declaration

```
void wait_for_next_period (void)

task_periodic_start_union  wait_for_next_period (void)
```

Argument

deadline_control:

0: Ok

1: missed at least one deadline.

Return codes

Nothing

Example

```
// Without deadline control
void t1(void)
{
```

```
    :
    init_period_time(50);

    while(1)
    {
        :
        :
      wait_for_next_period();
    }
}
…
// With deadline control
  task_periodic_start_union test;

    while(1)
    {
        :
        :
    test = wait_for_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 that is as long as there are 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:

sem_take
sem_release
sem_read

sem_take

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 sem_take (int semID)
```

Argument

semID	Semaphore number (0-15)

Return codes

Nothing

Example

```
#define SEM1 1

void t1(void)
{
```

```
    while(1)
    {
        :
        sem_take(SEM1); /* Pend on semaphore 1 */
        :
    }
}
```

sem_release

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 sem_release (int semID)`

Argument

semID	Semaphore number

Return codes

Nothing

Example

```
#define SEM1 1

void t1(void)
{

    while(1)
    {
        :
        sem_release(SEM1); /* Release semaphore 1 */
        :
    }

}
```

sem_read

Description

Read a task's semaphore status.

Function declaration

```
Sem_info_t sem_read (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 4 below.

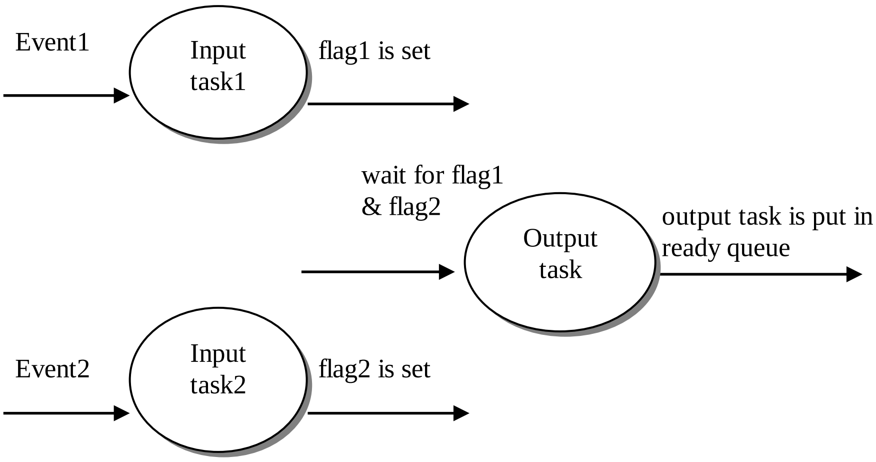

Figure 8 Flag example

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. Later on, task1 runs and sets flag1. In this scenario the output task will not be made ready at this point, as it asks for an *AND* operation between flag1 and flag2. After 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:

flag_wait
flag_set
flag_clear

flag_wait

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. Otherwise it will be suspended until the combination is set.

Function declaration

```
void flag_wait (int flag_mask)
```

Argument

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

Return codes

Nothing

Example

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

void t1(void)
{

   while(1)
   {
      :
      flag_wait(FLAG_MASK); /* Wait for Flag1 and
                               Flag3 to be set */
      :
   }

}
```

flag_set

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 flag_set (int flag_mask)
```

Argument

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

Return codes

Nothing

Example

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

void t1(void)
{

   while(1)
   {
      :
      flag_set(FLAG_MASK);   /* Set Flag1, Flag2 and
                                Flag3 */
      :
   }
}
```

flag_clear

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 flag_clear (int flag_mask)
```

Argument

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

Return codes

Nothing

Example

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

void t1(void)
{

   while(1)
   {
      :
      flag_clear(FLAG_MASK);  /* Clear Flag1, Flag2
                                 and Flag3 */
      :
   }
}
```

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.

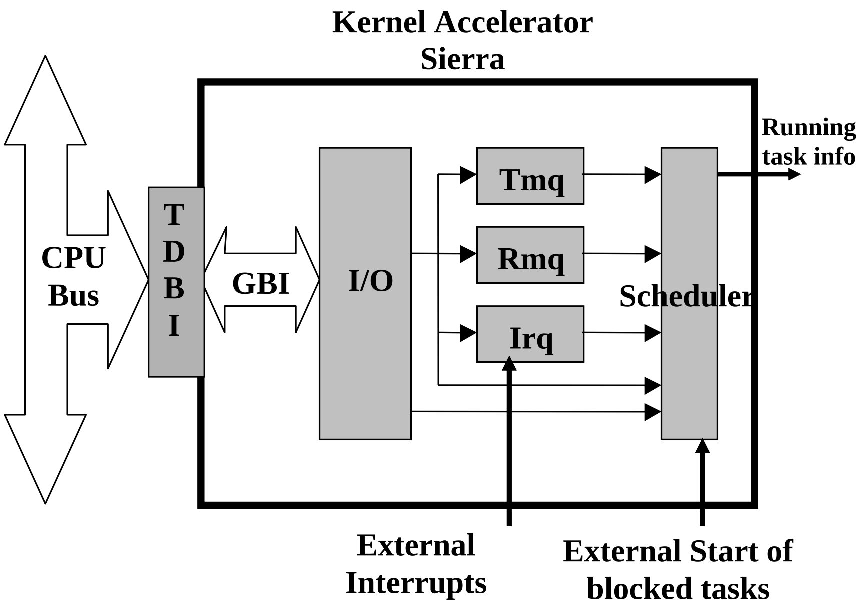

Figure 9: Block schematic

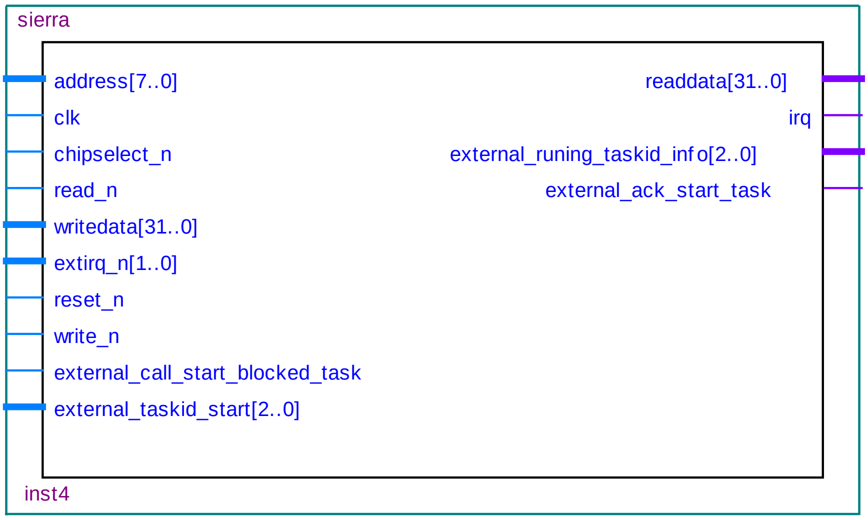

Figure 10: Sierra pin Interfaces

Table 2 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 API SW consists of the following files:

🗀 Sierra_RTOS
- 🗀 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
  - 🗀 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 debugg service calls)
    - alt_exception_enty.S (NiosII exception handling)
    - alt_exception_trap.S (NiosII trap handling)
    - component.mk (makefile)