Sierra 16

Stand-alone Real-Time Kernel in Hardware

Users Reference Manual
MicroBlaze™ edition

Version 0509

The Sierra 16 supports:

- 16 Tasks at 8 priority levels
- 16 Binary Semaphores
- 4 Flags
- 8 External Interrupts
- Timers - Delay, Watchdogs, Periodic tasks

COPYRIGHT INFORMATION

FURTHER INFORMATION

>> Further information about the Sierra RTOS can be found at www.realfast.se/sierra

CONTACT & SUPPORT

RealFast Intellectual Property
Vasteras Technology Park

Kopparbergsvagen 8
S – 722 13 Vasteras
Sweden

Email: susanna.nordstrom@realfast.se
Phone: +46 (0)21 – 470 20 25
Fax: +46 (0)21 – 470 21 25
URL: www.realfast.se/ierra

Table of Content

1 Purpose

The purpose of this users reference manual is to give system designers that are using the Sierra 16 operating system an overview and a functional description of how it works. Some examples are also included in this document as a help to getting started.

1.1 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 witch 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.

2 General Sierra 16 description

This chapter gives short overview of the Sierra 16 functionality. The Sierra 16 HW core is partitioned into modules as shown in the figure and described in the text below.

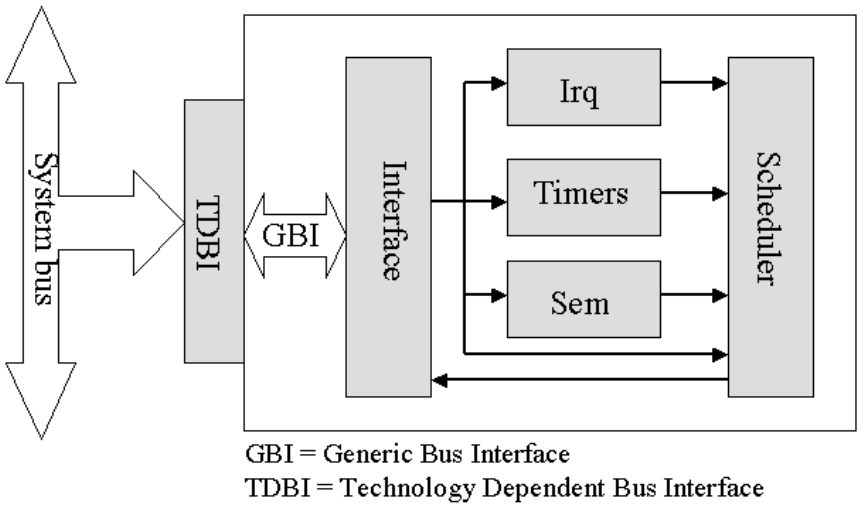


Figure 1: Overview of internal blocks in Sierra 16

2.1 Core Engine

Sierra 16 is partitioned into these functional units:

- Interface
- Scheduler
- Interrupt Controller
- Semaphore and Flag Handler
- Time Management Controller

The interface to Sierra 16 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 16 makes it very easy to interface it towards different busses.

2.2 Sierra 16 Interface

All communication (service calls) with the Sierra 16 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.

2.3 Scheduler

The Scheduler unit controls all scheduling in the Sierra 16. The scheduler can handle 16 tasks at 8 priority levels.

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 16.

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 16 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 (task_yield)
2. The task tries to lock a semaphore that is already locked and the task becomes blocked.
3. A task with a higher priority is becoming ready and therefore pre-empts the task and thereby placing the running task back into the ready-queue.

The Sierra 16 supports the following task management functions:

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

2.3.1 Task states

The Sierra 16 supports the following task states:

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

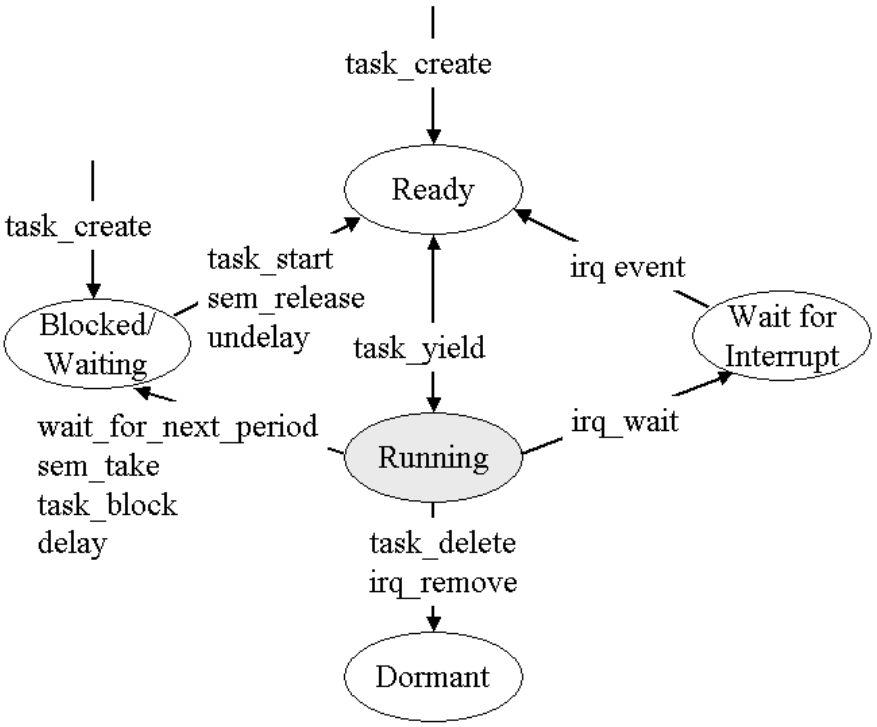


Figure 2: State transitions in Sierra 16

2.4 Interrupt Controller

The response time for interrupts, generated by external devices, must be kept short in all kinds of systems to obtain successful interaction with the external environment. Normally the external interrupts are connected to the CPU through some kind of interrupt controller chip, which means that when an external interrupt occur, the task that is for the moment running will be interrupted. This has of course effects on the predictability of the system, as an Interrupt Service Routine (ISR) will always run before a scheduled task.

By using the Sierra 16 intelligent interrupt handler a systems performance and behavior will improve, as each ISR is treated as a normal task. This is a big advantage that the Sierra 16 allows all tasks and ISRs to be scheduled. An ISR that needs short response time should be given a high priority. From a programmers point of view the handling of an external interrupt will almost the same as for an event flag.

The first thing to do in an ISR-task is to make a service call to the Sierra 16 to tell that this is an ISR-task and that it is waiting for an external interrupt. The Sierra 16 then will move the ISR-task from the running state to the 'Wait for irq-state'. When an interrupt occurs the ISR is scheduled by the Sierra 16 and will start to execute when it has the highest priority in the ready queue.

This type of interrupts handling increases the intelligence in the system while it is possible to prioritize a non-interrupt task before an interrupt task.

The Sierra 16 provides the following interrupt handling functions:

- turn on interrupts
- initialize an interrupt task
- wait for interrupt
- remove interrupt
- read status for interrupt

2.5 Semaphore and Flag Handler

The Sierra supports the use of 16 binary semaphores and 4 flags as synchronizing functions. Semaphores are used to synchronize resources in the system that are shared between tasks. A resource can be a 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 tasks are 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.

Note: There are totally 16 semaphores and flags available. This means that if you are using all 4 flags, there are 12 semaphores available.

These semaphore functions are supported:

- sem_take
- sem_release
- sem_read

These flag handling functions are supported:

- flag_wait
- flag_set
- flag_clear

2.6 Time Management Controller

In the Sierra there are three time-handling functions implemented, delay, support for periodic tasks and watchdog functionality. 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
- re-start of period
- delay/undelay
- wd_period_time
- wd_start
- wd_kick
- wd_stop

3 Sierra 16 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 SW that comes with the package is a driver for communication between CPU and HW kernel.

The Sierra handles:

- 16 tasks
- 8 priority levels
- 8 external interrupts
- 16 semaphores
- 4 flags
- Timers for delay, periodic tasks and watchdogs

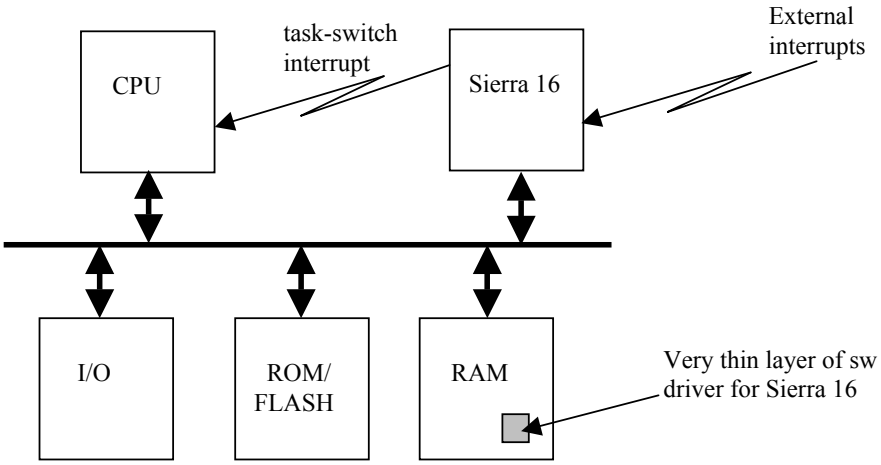


Figure 3: Example of system configuration

4 Task Management

This section describes the task handling services provided by the scheduler in 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 functions are implemented in the Sierra hardware kernel:

- Dynamic creation and removal of tasks (task_create, task_delete)
- Starting and blocking of tasks (task_start, task_block)
- Yield (task_yield)
- Get task status (task_getinfo)

4.1 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.

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 0-15). An idle task **must** be created and this task **shall** have taskID=0.
Priority	Specifies the priority of the task Range 0-7, where 0 is the highest-priority level. Priority level 7 is **reserved** for the idle task in the system.
taskstate	0 = task is initialized to the **blocked** state 1 = task is initialized to the **ready** state
taskptr	Pointer to code start for the task
stackptr	Pointer to task stack
stacksz	Size of the stack

RETURN CODES

Nothing

EXAMPLE

```
:
#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);
   :
}
```

4.2 task_delete

DESCRIPTION

Deletes the currently running task. This call can only be used to terminate the currently executing task. It is **not allowed** to perform this call from the idle task.

NOTE! Sierra does not de-allocate any potentially occupied semaphores etc. A task that has allocated a semaphore is responsible for de-allocation of it before the task is deleted.

FUNCTION DECLARATION

*void **task_delete** (void);*

ARGUMENT

N/A

RETURN CODES

Nothing

EXAMPLE

```
void t2(void)
{
   int i=0;

   while(i<25)
   {

      do something;
      i++;

   }

   task_delete(); /* After 25 turns in the loop, the task */
                  /* will delete itself. */
}
```

4.3 task_start

DESCRIPTION

Starts a task that is placed in the blocked state (un-block the task). The task will be moved from blocked state to the ready state.

FUNCTION DECLARATION

*void **task_start** (int taskId)*

ARGUMENT

task ID	Specifies the ID of the task (range 0-15)

RETURN CODES

Nothing

EXAMPLE

```
:
#define T2 2
:

void t1(void)
{

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

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

4.4 task_block

DESCRIPTION

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

FUNCTION DECLARATION

*void **task_block** (void)*

ARGUMENT

N/A

RETURN CODES

Nothing

EXAMPLE

```
void t1(void)
{
   int i=0;

   while(1)
   {

      i++;
      if(i==10)
         task_block(); /* Block t1 when i==10 */

   }
}
```

4.5 task_yield

DESCRIPTION

Yield to a task on the same priority if there is one, i.e. manual taskswitch. Otherwise the task will continue to run.

NOTE! Do not use this call in the startup sequence as it may lead to unpredictable results.

FUNCTION DECLARATION

*void **task_yield** (void)*

ARGUMENT

N/A

RETURN CODES

Nothing

EXAMPLE

```
void t1(void)
{

   while(1)
   {
        :
        :
        task_yield();  /* Yield to another task */

   }
}
```

4.6 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 0-15)

RETURN CODES

task_info_t	state_info (2 bits) : 0=Running, 1=Blocked, 2=Ready, 3=Dormant priority (3 bits) : 7 is the lowest priority level and 0 is the highest

EXAMPLE

```
:
#define T2 2
#define RUNNING 0
#define BLOCKED 1
#define READY   2
#define DORMANT 3
:

void t1(void)
{
   task_info_t info;

   while(1)
   {
      info = task_getinfo(T2);

      switch(info.state_info) /* info.state_info for task state */
      {                       /* info.priority for task priority */
          case RUNNING:
                :
          break;
          case BLOCKED:
                :
          break;
          case READY:
                :
          break;
          case DORMANT:
                :
          break;
      }
   }
}
```

4.7 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   taskswitch
                 interrupts*/
    :
    :
  }

}
```

4.8 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 taskswitch interrupts*/
    :
  }
}
```

4.9 set_timebase

DESCRIPTION

Sets the internal clock-tick timebase for the Sierra chip. By default the system is set to 100us clock-tick and the default clockspeed for calculating the ticks is 20MHz.

Examples of how to calculate the value of the time base register to obtain desired tick time.

1) Wanted tick time := 100µs
Clockspeed to Sierra := 20MHz

Timebase register value := 100µs * 20MHz/1000 => $100*10^{-6}*20*10^{6}/1000 := 2_{10} := 2_{16}$

2) Wanted tick time := 1ms
Clockspeed to Sierra := 50MHz

Timebase register value := 1ms * 50MHz/1000 => $1*10^{-3}*50*10^{6}/1000 := 50_{10} := 32_{16}$

3) Wanted tick time := 10ms
Clockspeed to Sierra := 100MHz

Timebase register value := 10ms * 100MHz/1000 => $10*10^{-3}*100*10^{6}/1000 := 1000_{10} := 3E8_{16}$

FUNCTION DECLARATION

*void **set_timebase** (int timebase)*

ARGUMENT

Timebase	Specifies the timebase of Sierra 16 (range 0-1023)

RETURN CODES

Nothing

EXAMPLE

```
void t1(void)
{

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

}
```

5 Interrupt Management

The section describes the functionality of the interrupt manager. The interrupts are associated with a interrupt-task which are scheduled as ”normal” tasks in the system. The following functions are implemented:

- irq_on_off
- irq_init
- irq_wait
- irq_remove
- irq_getinfo

If several interrupts occur simultaneously, the task associated with interrupt level 7 will be the first one sent to the ready queue.

5.1 irq_on_off

DESCRIPTION

This function is used to enable or disable the external interrupts connected to the Sierra HW kernel. All interrupts that are connected to Sierra are affected by this call.

FUNCTION DECLARATION

*void **irq_on_off** (int on_off)*

ARGUMENT

on_off	0 = Disable external interrupts, 1 = Enable external interrupts

RETURN CODES

Nothing

EXAMPLE

```
#define IRQ_DISABLE 0
#define IRQ_ENABLE  1

void t1(void)
{
  while(1)
  {
    :
    irq_on_off(IRQ_ENABLE);/*Enable external interrupt handling in Sierra*/
    :
  }
}
```

5.2 irq_init

DESCRIPTION

Initializes a specified interrupt to Sierra. Each interrupt pin can be programmed to be level or edge sensitive and also high/low level or rising/falling edge.

FUNCTION DECLARATION

void ***irq_init*** *(int irq, int edge_level, int hi_lo)*

ARGUMENT

irq nr	Interrupt number to initialize (range 0-7).
edge_level	1 = Edge sensitive, 0 = Level sensitive
hi_lo	If edge sensitive (edge_level = 1) falling edge = 0, rising edge = 1 If level sensitive (edge_level = 0) low level = 0, high level = 1

RETURN CODES

Nothing

EXAMPLE

```
#define IRQ_0  0
   :
#define IRQ_7  7

#define LEVEL  0
#define EDGE   1
#define LOW    0
#define HIGH   1


void t1(void)
{

  :
  irq_init(IRQ_0, EDGE, LOW); /*Initialize irq 0 to trig on falling edge*/
  irq_init(IRQ_7, LEVEL, HIGH); /*Initialize irq 1 to trig on high level*/
  :

}
```

5.3 irq_wait

DESCRIPTION

This call pends the calling task on the specified interrupt. The task is blocked until the interrupt occur (in wait_for_interrupt-state). When the interrupt occur, the task is transferred to the ready state and will start to execute when it has the highest priority in the ready state.

A task can be registered on more than one interrupt level, but each interrupt level can only support one task.

FUNCTION DECLARATION

*void **irq_wait** (int irq)*

ARGUMENT

irq_nr	Specifies the interrupt this task shall wait for (range 0-7)

RETURN CODES

Nothing

When this call is perfomed, the calling task will be blocked until the interrupt occurs. This means that the system needs at least one other task to switch to as the calling task is swaped out. So, at least an idle task is needed in the system when a task is using this call.

EXAMPLE

```
#define IRQ_0  0

void t1(void)
{

  while(1)
  {
    irq_wait(IRQ_0); /* Wait for irq 0 to occur */

     :
     :

  }
}
```

5.4 irq_remove

DESCRIPTION

Remove specified task from the ”wait for interrupt” state. The irq_remove call can be made at any point in time. To make the task ready to serve an interrupt again, the irq_wait function is called.

FUNCTION DECLARATION

*void **irq_remove** (int irq)*

ARGUMENT

irq	Interrupt level (range 0-7)

RETURN CODES

Nothing

EXAMPLE

```
#define IRQ_0  0

void t1(void)
{

  while(1)
  {
     :
    irq_remove(IRQ_0); /* Stop a task from waiting for irq 0 */
     :

  }
}
```

5.5 irq_getinfo

DESCRIPTION

Obtain status information on the specified interrupt level. The irq_getinfo call can be made at any point in time. The status of the specified interrupt level is returned in a data stucture of type *irq_info_t* with member variables as in the table below.

FUNCTION DECLARATION

irq_info_t ***irq_getinfo*** *(int irq_nr)*

ARGUMENT

irq_nr	Interrupt level (range 0-7)

RETURN CODES

irq_info_t

tasked	The task that is registered for this interrupt
edge_level	0=level, 1=edge
hi_lo	When Level; 0=low, 1=high, When edge; 0=falling, 1=rising
waiting	0=task is not waiting, 1=task is waiting
init	0=Interrupt is not initialized, 1=Interrupt is initialized

EXAMPLE

```
#define IRQ_3 3


void t1(void)
{
   irq_info_t info;
   int taskID, edge_level, hi_lo, waiting, init;


   while(1)
   {

      info = irq_getinfo(IRQ_3); /* Read status of irq 3 */

      /* The different member variables in the returned data-structure */
      taskID   = info.taskID;
      edge_level = info.edge_level;
      hi_lo      = info.hi_lo;
      waiting    = info.waiting;
      init       = info.init;
       :
       :
   }

}
```

6 Time Management

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

- delay
- undelay
- init_period_time
- wait_for_next_period
- restart_period
- read_timeq

6.1 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 is 1023.

RETURN CODES

Nothing

EXAMPLE

```
void t1(void)
{
   :

   while(1)
   {

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

}
```

6.2 undelay

DESCRIPTION

Remove the specified task from the timer queue (from blocked-state) and place it in the ready queue. The task will start to execute when it has the highest priority in the ready queue.

FUNCTION DECLARATION

void ***undelay*** *(int taskID)*

ARGUMENT

tasked	Task to be removed from the time queue and placed in the ready queue.

RETURN CODES

Nothing

EXAMPLE

```
#define T2 2

void t1(void)
{
   :

   while(1)
   {

      :
      undelay(T2); /* T2 is removed from the timer and inserted in
      :               the ready queue in Sierra */

   }

}
```

6.3 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*.

FUNCTION DECLARATION

void ***init_period_time*** *(int period_time)*

ARGUMENT

Period_time	Specifies the period time, in number of ticks, for the calling task. **Max value is 1023.**

RETURN CODES

Nothing

EXAMPLE)

```
void t1(void)
{
   :

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


   while(1)
   {

      :
      :

   }

}
```

6.4 read_timeq

DESCRIPTION

Return the actual period time or delay time left for the specified task.

FUNCTION DECLARATION

timeq_info_t ***read_timeq*** *(int taskID)*

ARGUMENT

tasked	The ID of the task to read the time queue of.

RETURN CODES

timeq_info_t

waiting	0 = No waiting task, 1 = Task is in timer queue
time	Ticks left of delay or periodtime

EXAMPLE

```
#define T2 2

void t1(void)
{
   timeq_info_t info;
   int time, waiting;

   while(1)
   {

      :

      info = read_timeq(T2);  /* Check timer for T2 */

      waiting = info.waiting; /* 0=not in timer queue, 1=in timer queue */
      time = info.time;       /* If task is in queue, time=ticks left */

      :

   }

}
```

6.5 wait_for_next_period

DESCRIPTION

Suspends a periodic task until the start of next period time.

FUNCTION DECLARATION

void ***wait_for_next_period*** *(void)*

ARGUMENT

N/A

RETURN CODES

Nothing

EXAMPLE

```
void t1(void)
{

   :
   init_period_time(50);

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

7 Watchdogs

In the Sierra there are support for watchdogs as each task has a timer in hardware that can be used for this function. Watchdogs are useful in systems where it is important to be able to handle and take care of errors that might occur during runtime. By having much of the functionality in the hardware kernel, the system becomes failsafe as the hardware kernel is independent of the SW and will always work regardless what happens in the SW - It simply cannot be overwritten.

The following functionality for watchdogs are available:

- wd_period_time
- wd_start
- wd_kick
- wd_stop

7.1 wd_period_time

DESCRIPTION

This call is performed in the watchdog to set the wakeup timer for the watchdog. This timer decides how long the watchdog will be put to sleep before waking it up if no other task kicks on it, in order to reset it. If the timer expires, the watchdog will be made ready and start to run according to its priority in the scheduler. Normally a watchdog should never wake up, as the associated timer is supposed to reset the watchdog on a regular time base by some task in the system.

Note - This call should be performed when the watchdog task is initializing itself.

FUNCTION DECLARATION

void **wd_period_time** (int wd_time)

ARGUMENT

wd_time	Specifies the time in ticks before the timer expires.

RETURN CODES

Nothing

EXAMPLE

```
void watchdog(void)
{
   :

   wd_period_time(700); /* Initialize timer for watchdog to 700 ticks */

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

7.2 wd_start

DESCRIPTION

This call starts the watchdog timer. The wd_period_time call must have been called before starting the timer. The watchdog task is suspended when this call is performed and it will only start if the watchdog timer expires or if the wd_stop call is performed.

FUNCTION DECLARATION

void **wd_start** (void)

ARGUMENT

N/A

RETURN CODES

Nothing

EXAMPLE

```
void watchdog(void)
{
   :

   wd_period_time(700); /* Initialize timer for watchdog to 700 ticks */

   while(1)
   {
      wd_start();       /* Put the watchdog task to sleep */
      :
      :
   }
}
```

7.3 wd_stop

DESCRIPTION

This call stops the watchdog timer and makes the watchdog task ready. It will run when it has the highest priority in the ready queue.

FUNCTION DECLARATION

void **wd_stop** (int taskID)

ARGUMENT

tasked	The task -id of the watchdog task

RETURN CODES

Nothing

EXAMPLE

```
#define WD_TASK 10            /* Watchdog task has id number 10*/


void task(void)
{
   :


   while(1)
   {
      :
      wd_stop(WD_TASK);       /* Stop the timer and make the wd ready */
      :
   }
}
```

7.4 wd_kick

DESCRIPTION

The wd_kick call resets the timer for the watchdog. This call must be performed on a regular time basis in order to prevent the watchdog task to start. The easiest way to do this is to have a periodic task in the system that makes this call. The period time of this task must of course be shorter than the watchdog timer period to be able to guarantee that the watchdog will not be activated.

FUNCTION DECLARATION

void **wd_kick** (int taskID)

ARGUMENT

tasked	The task -id of the watchdog task

RETURN CODES

Nothing

EXAMPLE

```
#define WD_TASK 10              /* Watchdog task has id number 10*/


void task(void)
{
   :

   while(1)
   {
      :
      wd_kick(WD_TASK);         /* Reset the timer for the watchdog */
      :
   }
}
```

8 Semaphore Management

This section describes the functionality of the semaphore management. There are 16 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 15 other tasks can be waiting for it. The queue is arranged in task-id numbers and the task with highest id-number in the queue will get the semaphore when it becomes available.

Note: There are totally 16 semaphores and flags available. This means that if you are using all 4 flags, there are 12 semaphores available.

- sem_take
- sem_release
- sem_read

8.1 sem_take

DESCRIPTION

Makes a task pending 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 queue for the semaphore until it becomes free.

Note: The queue is arranged in task-id numbers and the 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 */
      :
   }
}
```

8.2 sem_release

DESCRIPTION

Releases the specified semaphore and if there are one or more tasks waiting for the semaphore, the first task in the queue will get the semaphore and it will be moved to the 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 */
      :
   }

}
```

8.3 sem_read

DESCRIPTION

Read the status of a semaphore.

FUNCTION DECLARATION

Sem_info_t ***sem_read*** *(int semID)*

ARGUMENT

semID	Semaphore to read status of.

RETURN CODES

Sem_info_t

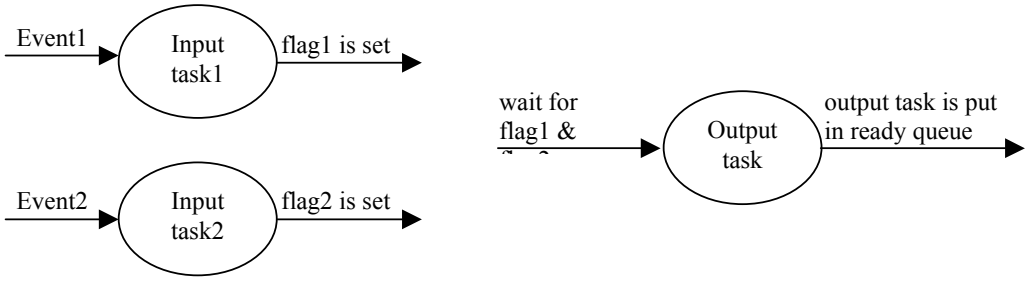
taskID	Waiting task id
status	0 = Free, 1 = Locked
waiting	0 = No task is waiting (ignore taskID), 1 = There is a task in the waiting queue

EXAMPLE

```
#define SEM3 3

void t1(void)
{
   sem_info_t sem;
   int taskID, status, waiting;

   while(1)
   {

      :
      sem = sem_read(SEM3);

      /* The different member variables in the returned data-structure */
      waiting = sem.waiting;
      status = sem.status;
      taskID = sem.taskID;

   }
}
```

9 Flags

The Sierra has support for flags for efficient synchronizing of events. The entire synchronizing algorithm is handled by the HW kernel. This makes the handling of the flags very efficient as 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 is a output task that these input tasks will trigger - see figure.



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 gets 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 a *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 rum when it has the highest priority in the ready queue.

The Sierra supports four flags that can be used in 2^4-1 (=15) different combinations.

The following flag handling functions are supported:

- flag_wait
- flag_set
- flag_clear

Note: There are totally 16 semaphores and flags available. This means that if you are using all 4 flags, there are 12 semaphores available.

9.1 flag_wait

DESCRIPTION

This call makes a task wait for one or more flags to be set. If the flag(s) are allready 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 */
      :
   }

}
```

9.2 flag_set

DESCRIPTION

This call sets one or more flags. If the 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 */
      :
   }
}
```

9.3 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 are 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 */
      :
   }
}
```

10 Startup sequence

When the system is booting and starts to run, there are a few things that need to be done to start up the Sierra operating system.

1. Enclosed in the MicroBlaze package, there is an interrupt controller for taking care of external interrupts. Together with this interrupt controller there are some routines for attaching functions to handle the different interrupts. However, with the Sierra the interrupt handling changes and therefore the 'default' routines are replaced by a routine enclosed in the Sierra package. In the example below, it is assumed that the interrupt controller is not used, i.e. the Sierra is connected directly to the interrupt line on the MicroBlaze CPU.
2. The Sierra operating system needs to be initialized before it can be used. There are some internal datastuctures etc. that needs to be initialized before it is ready to use.
3. The scheduling starts by enabling the task-switch interrupts from Sierra.

EXAMPLE

```
int main (void)
{
  /* Override default interrupt handler */
  set_irq_handler();

  /* Enable the interrupts */
  microblaze_enable_interrupts();

  /*******************************************************
    Initialize time base register.
    In this example  : 24MHz system clock
    Wanted tick time : 1ms
    Formula gives    : 1ms x 24MHz /1000 => 18 (hex)
  *******************************************************/

  Sierra_Time_base_reg=0x18;

  /* Initialize OS internals */
  os_init();

  /* TaskID=1, Prio=1, State=ready*/
  task_create(T1, 1, READY, t1, stack1, STACK1_SZ);

  /* TaskID=2, Prio=,2 State=blocked*/
  task_create(T2, 2, BLOCKED, t2, stack2, STACK2_SZ);

  /* TaskID=15, Prio=7, State=ready*/
  task_create(IDLE, 7, READY, idle, idle_stack, IDLESTACK_SZ);

  /* Start the scheduling */
  tsw_on();

  while(1) /* Should never happen... */
     print ("Error in startup...\n\r");
  return(0);
}
```

10.1 API - The SW files

There are a number of SW files in the Sierra package. By intention the SW has been kept small, as the idea with the Sierra is to get a sophisticated real time system with smallest possible footprint.

In this section the API (Application Programmer Interface), i.e. the SW driver files are described. The driver is written with the possibility for the user to configure it so it will suit your specific needs. In some cases you might want to use all kinds of functionality in the Sierra and in some cases maybe only parts of all functionality.

Below some of the most important files are described so you easily will find things you might need to edit.

10.1.1 Sierra.c

This is the file where the major part of the driver code is implemented. We recommend not to do any changes to this code as it is well tested and verified.

10.1.2 Sierra_base_adr.h

This file imports a generated base address from the Xilinx Platform Studio toolkit. The base address is found in a file namned `xparameters.h` and the base address is named `XPAR_Sierra_1_BASEADDR`. In `Sierra_base_adr.h` this constant is renamed to `Sierra_BASE_ADR`. If, for some reason, the name of the generated and imported base address should be changed, you need to change the name in this file as well.

10.1.3 Sierra_io.h

In this file, datastructures and encoding of system calls are defined.

10.1.4 Sierra_ker.h

If you want to change the configuration of the driver you can do it in this file. Here you can define how many tasks the system shall support and here you can also include or exclude functionality.

Important constants:

```
#define N_TASKS 16
```

N_TASKS is used to allocate memory area for each tasks tcb. If you need fewer tasks than the default value 16, you will save space in memory if you decrease this value.

By `define` or `undef` these constants in `Sierra_ker.h`, you can decide which functions in the Sierra the driver shall support.

```
#define TIMERS
/*#undef TIMER*/

#define EXT_INTERRUPTS
/*#undef EXT_INTERRUPTS*/


#define SEMAPHORES
/*#undef SEMAPHORES*/

#define FLAGS
/*#undef FLAGS*/
```

By default all functionality is included, but if you for any reason is not going to use some functions, you can undef the constant to minimize the memory footprint of the driver.

10.1.5 Sierra_reg.h

Constants for addresses to all registers in the Sierra are defined in this file.

10.1.6 Tcb.h

This is the tcb (Task Control Block) structure that contains status for each task in the system. It is mainly the CPU registers and stack pointers that are stored in the tcb. Other task status data is stored internally in the Sierra hardware.

10.1.7 tcb_offset.h

Offsets to the different areas in the tcb are defined in this file. The different registers and their usage in the MicroBlaze CPU is also described in this file.

10.1.8 csw.s

This is the assembler routines for saving-restoring the registers in the MicroBlaze when a task is swapped out and replaced by another task.

10.1.9 main.c

The `main.c` file contains a simple example of how to use the Sierra. In this example three tasks are created, one idle task and two application tasks. By looking at this example, you probably will get the basic idea of how to build a multitasking real time system by using the functions in the Sierra.

See example code in next page.

```
/*********************************************************************

   COPYRIGHT (C) 2002 RealFast

   All rights reserved. RealFast's source code is an unpublished work,
   and the use of a copyright notice does not imply otherwise. This
   source code contains confidential, trade-secret material of RealFast.
   Any attempt at or participation in deciphering, decoding, reverse
   engineering, or in any way altering the source code is strictly
   prohibited unless the prior written consent of RealFast is obtained.

*********************************************************************/

/*********************************************************************
    This file contains an example of how to use the Sierra HW-RTOS.
    The intention with the example is to make it easy to get started with
    the MicroBlaze-Sierra system.
*********************************************************************/


#include <mb_interface.h>
#include <sierra_reg.h>
#include <sierra_ker.h>
#include <sierra_io.h>

extern  void set_irq_handler(void);
extern  void print(char *);

/* Task id´s */
#define T1 1
#define T2 2
#define IDLE 0

/* States */
#define BLOCKED 0
#define READY 1

/* Semaphores */
#define PRINT_SEM 1

/* Task stacks */
char stack1[200];
char stack2[200];
char idle_stack[200];
```

```
/**** Task 1-2 below ***********************/
void t1(void)
{
   task_start(T2);

   while(1)
   {

      sem_take(PRINT_SEM);
      print("Task1 is alive...\n\r");
      sem_release(PRINT_SEM);
      delay(100);
   }

   /* Should never end up here... */
   task_delete();
}

void t2(void)
{
   init_period_time(900);

   while(1)
   {

      sem_take(PRINT_SEM);
      print("Task2 is also alive!!!!\n\r");
      sem_release(PRINT_SEM);
      wait_for_next_period();

   }

}
/*****************************************/


/*** The idle task ************************/
void idle(void)
{
   int i=0;

   print ("******************\n\r");
   print ("*Idle starts Task1*\n\r");
   print ("******************\n\r");

   task_start(T1);
   while(1)
   {
        for(i=0;i<500000;i++);
   };

}
/*****************************************/
```

```
int main (void)
{
  /* Override default interrupt handler  */
  set_irq_handler();
  microblaze_enable_interrupts();

  /************************************************************
     Initialize time base register.
     This example     : 24MHz system-clock
     Wanted tick time : 1ms
     Formula gives    : 1ms x 24MHz / 1000 => 24(dec) or 18(hex)
   ************************************************************/

  Sierra_Time_base_reg=0x18;
  os_init();

  /* TaskID=1, Prio=1, State=block */
  task_create(T1, 1, BLOCKED, t1, stack1, 200);

  /* TaskID=2, Prio=1, State=block */
  task_create(T2, 1, BLOCKED, t2, stack2, 200);

  /* TaskID=0, PRIO=7, State=ready */
  task_create(IDLE, 7, READY, idle, idle_stack, 200);

  /* Start the scheduler */
  tsw_on();

  while(1)
     print ("Shouldn't bee here!!!!\n\r");
  return(0);
}
```