Running uC/OS on the X32 Soft Core

M. Dufour, S. Woutersen, A.J.C. van Gemund

Embedded Software Lab Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

March 2006

1   Introduction

We describe how to use uC/OS [2], a simple, preemptive, multitasking RTOS, in combination with the X32 soft core [1]. We also describe how we have modified uC/OS and the X32 to operate together.

2   Using uC/OS

In this section, we provide an overview of the most important uC/OS functions and examples of their usage. We also describe how to create interrupt service routines in an uC/OS application.

2.1   Overview

The function `OSInit` should always be called first. Besides initializing the uC/OS datastructures, this function creates an *idle task*, that is run when there are no other ready tasks. Any tasks, semaphores, etc. should be created after calling `OSInit`. To actually start multitasking, the function `OSStart` is used to switch to the highest-priority ready task.

The function `OSTaskCreate` is used to create up to 62 tasks, each of which should have a *different priority*. These limitations allow uC/OS to quickly determine the highest-priority ready task at any time. The following shows the arguments that are passed to `OSTaskCreate`:

```
OSTaskCreate(func_name, task_arg, stack_space, priority);
```

The first argument is the name of a function that implements the task's behaviour. The second argument allows for an initial argument to be passed to the function, so that multiple tasks can be implemented with a single function. The third argument points to some preallocated memory space, to be used as the task's working stack. The final argument is the task's priority, ranging from 1 to 62 (63 is used for the idle task). The *lower* the number, the *higher* the priority.

Each *time tick* (an interrupt generated by a hardware timer at a fixed time interval), uC/OS determines the highest-priority ready task. If this is not the current task, it *preempts* the current task, and performs a *context switch*. Tasks can also yield 'voluntarily', by calling the `OSTimeDly` function (a single argument determines after how many time ticks the task should become ready again), or by blocking on some resource (e.g., a queue or something protected by a semaphore).

The following shows a minimal example of how the mentioned functions can be used together. Two tasks are created, that print their initial argument at each time tick, in the order of their priorities:

```
#include <ucos.h>

void *task1_stack[1024];
void *task2_stack[1024];
```

1

```
void task(void *arg) {
    while(TRUE) {
        printf(''%d\n'', (int)arg);
        OSTimeDly(1);
    }
}

int main() {
    OSInit();
    OSTaskCreate(task, (void *)8, task1_stack, 1);
    OSTaskCreate(task, (void *)9, task2_stack, 2);
    /* insert other initialization code here */
    OSStart();
}
```

This example repeatedly outputs "89", because the task with argument 8 has a higher priority than the task with argument 9. (Note that uC/OS works with `void *` types in many places, so we have to cast objects with other types.)

uC/OS provides basic support for *semaphores*, *queues* and *mailboxes*. For each of these, it additionally provides a *non-blocking* function to check their state.

To create a *semaphore*, the function `OSSemCreate` is used. A single argument denotes its initial counter value. Therefore, an argument of 1 can be used to create a binary semaphore. To pend on a semaphore, the function `OSSemPend` is used. One of its arguments is a *timeout* value, used to unblock the task after a certain number of time ticks. Setting it to WAIT_FOREVER (defined as 0) disables the timeout. If a timeout occurs, this is indicated via the third argument. To post to a semaphore, the function `OSSemPost` is used. These functions respectively increment and decrement the semaphore counter. The non-blocking version of `OSSemPend` is called `OSSemAccept`. If the counter is 0, the resource protected by the semaphore is not available, so `OSSemAccept` returns 0. Otherwise, it *decrements* the counter and returns its value[1] . This means that `OSSemPost` should be called to increment the counter once more.

The following example shows how to use these functions to create and use a binary semaphore:

```
#include <ucos.h>

void *task_stack[1024];

UBYTE err;         /* used to store error codes */
OS_EVENT *pSem;    /* pointer to semaphore object */

void task(void *arg) {
    while(TRUE) {
        OSSemPend(pSem, WAIT_FOREVER, &err);
        printf(''got the semaphore once\n'');
        OSSemPost(pSem);

        if(OSSemAccept(pSem)) {
            printf(''got the semaphore twice\n'');
            OSSemPost(pSem);    /* increment counter */
```

---

[1]This instruction is akin to the test-and-set instruction, sometimes found at the hardware level, which atomically tests a bit value, and sets the bit if it was 0.

```
        }

        OSTimeDly(1);
    }
}

int main() {
    OSInit();
    OSTaskCreate(task, 0, task_stack, 1);
    pSem = OSSemCreate(1);    /* 1 means binary semaphore */
    OSStart();
}
```

To create a *mailbox*, the function `OSMboxCreate` is used. In uC/OS, mailboxes can only contain a *single* message. The argument of `OSMboxCreate` denotes the initial message. If it is 0, the mailbox is initially empty. `OSMboxPost` is used to post a message. `OSMboxPend` and `OSMboxAccept`, like the corresponding semaphore functions, are used to pend on or check the state of the mailbox. When available, they return the current message.

To create a *queue*, the function `OSQCreate` is used. It accepts two arguments, that should point to preallocated space for the queue, and its size. To pend on, post and check the state of a queue, similar functions as for mailboxes are used, called `OSQPost`, `OSQPend` and `OSQAccept`. The following example shows how to create a queue.

```
#define queue_size 32
void *queue[queue_size];
...
OSQCreate(queue, queue_size);
```

2.2 Interrupts

In a typical hardware setup, there are multiple interrupt sources, such as hardware timers, UART's and I/O lines, each with an associated, hardware dependent *interrupt number*. Typically, these numbers are unique; if not, the respective interrupt handler either does not know or may have to spend cycles to find out the specific type of interrupt.

In real-time systems, certain interrupts are also typically more time-critical than others. For example, when dealing with in- and output, data often has to be read or written quickly, or it may be overwritten. Therefore, each interrupt (number) is typically associated with a user-defined *priority*. Interrupt service routines are *preempted* whenever a higher-priority interrupt occurs, and resumed when all higher-priority interrupts have been handled. In a way, this makes them similar to tasks, but with even higher priorities.

Aimed at providing multitasking capabilities, uC/OS has only rudimentary support for dealing with interrupts. It does not provide interrupt handling services, such as declaring interrupt handlers. This means the low-level X32 library [1] must be used for this purpose. The following shows an example of how to setup an *interrupt service routine* (ISR).

```
#include <x32.h>

SET_INTERRUPT_VECTOR(INTERRUPT_BUTTONS, isr_buttons);
SET_INTERRUPT_PRIORITY(INTERRUPT_BUTTONS, 10);
ENABLE_INTERRUPT(INTERRUPT_BUTTONS);
```

For this to work, there must be function named `isr_buttons`, with no arguments or return type. We assume `INTERRUPT_BUTTONS` to be a predefined interrupt number (in our setup, it is defined in `x32.h`). The priority can be practically any number larger than 0, but for portability sake, we suggest using a number from 1 to 15. In contrast to uC/OS priorities, a higher number here means a higher priority![2].

Because interrupt service routines may wake up a high-priority task, or cause the current task to be suspended, uC/OS needs to know when all interrupts have been handled, so that it can determine if a context switch is necessary. This is achieved by keeping track of *interrupt nesting*, that may occur because of ISR preemption: if the nesting depth becomes 0, all interrupts have been handled. uC/OS provides two functions to keep track of interrupt nesting, called `OSIntEnter` and `OSIntExit`. These functions must be called by the user on entry and exit of each ISR, respectively.

The following example shows how to create an interrupt service routine to count the number of interrupts, caused by pressing some button. To be able to respond quickly, it delegates the job of printing the counter to a separate task[3].

```
#include <ucos.h>
#include <x32.h>

void *task_stack[1024];

#define queue_size 32
void *queue_mem[queue_size];

UBYTE err;
OS_EVENT *queue;

int counter = 0;

void isr_buttons() {
    OSIntEnter();    /* should always be called on entry */
    OSQPost(queue, (void *)counter++);
    OSIntExit();     /* should always be called on return */
}

void task(void *arg) {
    int i;
    while(TRUE) {
        i = (int)OSQPend(queue, WAIT_FOREVER, &err);
        printf(''%d\n'', i);
    }
}

int main() {
    OSInit();

    SET_INTERRUPT_VECTOR(INTERRUPT_BUTTONS, isr_buttons);
    SET_INTERRUPT_PRIORITY(INTERRUPT_BUTTONS, 10);
    ENABLE_INTERRUPT(INTERRUPT_BUTTONS);
```

---

[2]The name 'setvect' originates from the concept of 'interrupt vector', a table in memory that describes the ISR (and in case of the X32, the priority) associated with each interrupt (number).

[3]Note that printing some characters, e.g., via a 'slow' RS232 connection takes a very large amount of time, when compared to executing individual instructions.

```
        OSTaskCreate(task, (void *)0, task_stack, 50);
        pQueue = OSQCreate(queue_mem, queue_size);

        OSStart();
    }
```

In Appendix A, we provide a reference description of the most important uC/OS functions.

3   Porting uC/OS to the X32

In this section, we describe how we have modified uC/OS and the X32 processor to operate together. First, we explain how we implement *context switching*. Second, we explain how *(prioritized) interrupts* are handled.

3.1   Context Switching

Context switching is typically implemented using the standard C `setjmp` and `longjmp` functions. The `setjmp` function saves the processor state (its registers) and the address of the top of the current working stack. The `longjmp` function restores these again, allowing us to resume a task from the point where `setjmp` was called.

The return value of `setjmp` is used to differentiate between a normal return and the result of a `longjmp` (and possibly between different calls to `longjmp`). If `setjmp` returns normally, the return value is 0. If it is resumed as a result of a `longjmp`, it returns the value given by the programmer as the argument to `longjmp`.

The following example shows how we can use the `setjmp` and `longjmp` functions to implement (cooperative) multitasking in uC/OS:

```
#include <setjmp.h>

void task(void *arg) { /* example task */
    while(TRUE) {
        OSTimeDly(1); /* this function calls the scheduler */
    }
}

void OSSched() { /* uC/OS scheduler */
    /* determine if we need to switch to a new task */
    ...

    /* if so:  perform context switch */
    if(setjmp(current_task->state) != 0)
        return; /* we came here via longjmp:  resume task */

    /* otherwise, jump to new task */
    longjmp(new_task->state, 1); /* causes setjmp to return 1 */
}
```
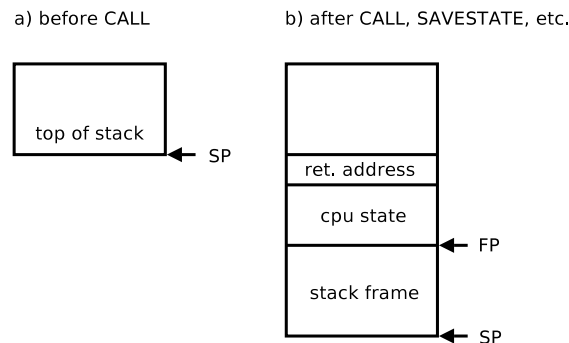
To enable preemption, we simply cause `OSSched` to be called each time tick, using a hardware timer interrupt. The respective interrupt service routine simply calls `OSSched`, using the current

working stack, so that the current task continues after it returns (i.e., when it is switched to later on).

The `setjmp` and `longjmp` functions are separate functions, because they can also be used for other purposes than multitasking. The non-local jumps they enable can, for example, also be used to implement features similar to exception-handling in C. (Consider a function jumping back to `main`, after some error condition has occurred.)

Because the X32 is stack-based, the stack may change between calling `setjmp` and `longjmp`, complicating their implementation. For our single purpose of context switching, we combine the two using a single function. It relies on the X32 `CALL`, `SAVESTATE` and `RET` instructions to save and restore processor state. (Most state is on the stack of course, but some additional registers are used to manage the stack.) It further uses the `LOADFP` and `SAVEFP` instructions to store and restore the top of the current working stack.

The following figure shows the stack before executing a `CALL` instruction, and after executing `CALL` and the first few instructions of the called function, that are always the same:

a) before CALL   b) after CALL, SAVESTATE, etc.

top of stack ← SP

ret. address
cpu state ← FP
stack frame ← SP

The `CALL` instruction pushes the return address on the stack. The `SAVESTATE` instruction next pushes the processor state on the stack. The following instructions create a local stack frame for the called function. The `SP` register always points to the top of the stack, and the `FP` register to the start of the local stack frame.

The `RET` instruction works as follows. First, it assigns `FP` to `SP`, in essence removing the local stack frame. Next, it pops the processor state and return address from the stack, and jumps to the return address.

Our context switch function simply saves `FP`, so that the `RET` instruction can later be used to resume the current task. In pseudo-code, the function looks like this (of course, the `RET` instruction is implicit here):

```
void context_switch(task *current_task, task *new_task) {
    current_task->fp = FP; /* save pointer to current state */
    FP = new_task->fp; /* prepare cont.sw.  through RET */
}
```

To switch to a task for the first time, we need to setup its stack and `FP` register such that the processor state is correctly initialized, i.e., the `RET` instruction pops the correct initial processor state, including the initial execution address, from the stack. We also need to place a task's initial argument on the stack. We setup the `FP` register and initial stack for each task in the uC/OS function `OSTCBInit`, that is called when a task is created.
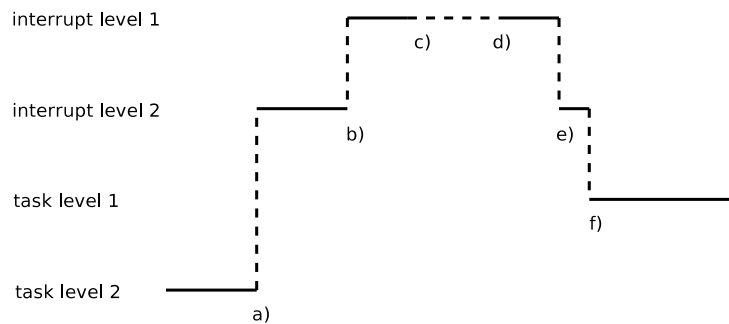
3.2 Interrupts

The work of handling (prioritized) interrupts must be somehow divided between hard- and software. One extreme approach is to do as much as possible in software. In this case, hardware is only used to notify the software when an interrupt occurs, by jumping to a general interrupt handler. The interrupt handler then resolves the correct interrupt number and priority, and keeps track of interrupt nesting. The result is a very simple, flexible, and (from a hardware perspective) cost-efficient solution. However, a software approach comes with certain overheads, that are especially undesirable in real-time systems. Consider, for example, several low-priority interrupts arriving, while servicing a high-priority interrupt: the interrupt handler is called several times, dramatically increasing the response time of the high-priority interrupt.

To be able to respond quickly to high-priority interrupts, we implement support for multiple interrupts and priorities in the X32 processor. Located at a fixed (virtual) memory address, an *interrupt vector* contains both the address and priority of each interrupt. At all times, the processor maintains an *interrupt level*, corresponding to the priority of the interrupt currently being serviced. New interrupts are either handled directly, if their priority is higher than the current level, or delayed until all higher-priority interrupts have been handled. If a new interrupt is to be handled, the processor raises the interrupt level to the respective priority and jumps to the respective address in the interrupt vector.

There are two types of *nesting issues*. First, we need to restore the interrupt level once an interrupt is handled. This is done (in hardware) by pushing and popping the interrupt level on/off the stack, for every (interrupt) `CALL` and `RET` instruction, respectively. Second, uC/OS must be able to perform a context switch, if necessary, to a different task than the current task, once all interrupts are handled, i.e., when the interrupt nesting depth becomes 0. To accomplish this, we require the programmer to manually add calls to `OSIntEnter` and `OSIntExit` on function entry and exit, respectively, as mentioned in Section 2.2.

Because an interrupt may be preempted by a higher-priority interrupt before `OSIntEnter` is called, the interrupt nesting depth may become 0, while there are still interrupts waiting to be serviced. To avoid performing a context switch in such cases, we modify `OSIntExit` to also look at the interrupt level on the stack, to see if other interrupts should be finished first.

The following figure illustrates how our combined architecture solves the aforementioned nesting issues, for a combination of different tasks and interrupts:

Let us walk through the different transitions indicated in the figure. While running a low-priority, level 2, task, a level 2 interrupt occurs (*transition a*). Because all tasks run at the lowest-possible interrupt level, the processor decides to service the interrupt. It places the current interrupt level on the stack, raises the interrupt level to 2 and passes control to the respective interrupt service routine. The interrupt service routine, in turn, calls `OSIntEnter`, so uC/OS knows the interrupt nesting depth is now 1.

While the level 2 interrupt is being serviced, a high-priority, level 1, interrupt occurs (*transition b*). The processor decides that it should preempt the interrupt that is being serviced. Again,

it places the current interrupt level (2) on the stack, and raises the current interrupt level to 1. Another call to `OSIntEnter` now increments the interrupt nesting depth to 2.

The current interrupt level is only increased when an interrupt service routine is first called. This means that, whenever an interrupt handler calls an uC/OS or user-defined function (*transition c*), the current interrupt level does not change. The current interrupt level is simply placed on the stack by the `CALL` instruction, and restored by the `RET` instruction (*transition d*).

When the level 1 priority interrupt is handled, its service routine calls `OSIntExit`, so uC/OS knows the interrupt nesting depth is again 1. Via the `RET` instruction, the processor now restores the state of the low-priority interrupt, including its interrupt level (*transition e*).

When the level 2 priority interrupt is handled, its service routine also calls `OSIntExit`, so the interrupt nesting depth is 0. Because it is 0, uC/OS now checks that there are no other interrupts waiting to be serviced and directly switches to the high-priority, level 1, priority task that happens to have become ready somewhere during interrupt servicing (*transition f*).

References

[1] Woutersen S., *X32 Resource Site*, `http://x32.swoutersen.nl`

[2] Labrosse J.J., *uC/OS Website*, `http://micrium.com`

A uC/OS Function Reference

In this appendix, we present a reference description of the most important uC/OS functions (in alphabetical order).

- `void OSInit(void)`

  Initialize uC/OS data structures, and create a low-priority idle task. This function should always be called, before calling any other uC/OS function!

- `void OSIntEnter(void)`

  Increment counter that maintains interrupt nesting depth. This function should always be called on entry of an interrupt service routine.

- `void OSIntExit(void)`

  Decrement counter that maintains interrupt nesting depth. If the counter becomes 0, check to see if all interrupts have been handled, and perform a context switch if necessary (e.g., because a high-priority task has become ready).

- `void *OSMboxAccept(OS_EVENT *pevent)`

  Determine if a message is available in mailbox `*pevent`, without suspending the current task if not (as opposed to `OSMboxPend`). Return the message if available and clear the mailbox. Otherwise, return 0.

- `OS_EVENT *OSMboxCreate(void *msg)`

  Create and return mailbox, with initial message `*msg` (0 means no initial message). An uC/OS mailbox can only contain a single message.

- `void *OSMboxPend(OS_EVENT *pevent, UWORD timeout, UBYTE *err)`

  Determine if a message is available in mailbox `*pevent`, while suspending the current task if not (as opposed to `OSMboxAccept`). If no message becomes available, make the current task ready again after `timeout` time ticks (wait forever if `timeout` is 0), set `*err` to `OS_TIMEOUT` and return 0. Otherwise, set `*err` to `OS_NO_ERR` and return the message.

- `UBYTE OSMboxPost(OS_EVENT *pevent, void *msg)`

  If there are tasks pending on mailbox `*pevent`, send `*msg` to the highest-priority task, switch to this task and (when the current task is resumed) return `OS_NO_ERR`. Otherwise, return `OS_MBOX_FULL` if there already is a message in the mailbox, or place the message in it, and return `OS_NO_ERR`.

- `void *OSQAccept(OS_EVENT *pevent)`

  Determine if a message is available in queue `*pevent`, without suspending the current task if not (as opposed to `OSQPend`). Return the message if available and remove it from the queue. Otherwise, return 0.

- `OS_EVENT *OSQCreate(void **start, UBYTE size)`

  Create and return queue of maximum size `size`. Memory should be preallocated at location `start`. The following example creates a queue of size 32:

  ```
  #define QSIZE 32
  void *queue[QSIZE];
  OS_EVENT *pqueue = OSQCreate(queue, QSIZE);
  ```

- `void *OSQPend(OS_EVENT *pevent, UWORD timeout, UBYTE *err)`

  Determine if a message is available in queue *pevent, while suspending the current task if not (as opposed to OSQAccept). If no message becomes available, make the current task ready again after timeout time ticks (wait forever if timeout is 0), set *err to OS_TIMEOUT and return 0. Otherwise, set *err to OS_NO_ERR and return the oldest message.

- `UBYTE OSQPost(OS_EVENT *pevent, void *msg)`

  If there are tasks pending on queue `*pevent`, send `*msg` to the highest-priority task, switch to this task and (when the current task is resumed) return `OS_NO_ERR`. Otherwise, return `OS_Q_FULL` if the queue has reached its maximum size, or place the message at the back of the queue, and return `OS_NO_ERR`.

- `UWORD OSSemAccept(OS_EVENT *pevent)`

  Return current counter of semaphore `*pevent` and decrement it, if greater than 0 (the task is not suspended, as opposed to `OSSemPend`). Otherwise, return 0.

- `OS_EVENT *OSSemCreate(UWORD cnt)`

  Create and return semaphore, with initial counter value `cnt`. Note that a value of 1 results in a binary semaphore.

- `void OSSemPend(OS_EVENT *pevent, UWORD timeout, UBYTE *err)`

If the counter of semaphore `*pevent` is greater than 0, decrement the counter, set `*err` to `OS_NO_ERR` and return. Otherwise, suspend the current task (as opposed to `OSSemAccept`). If the counter does not become greater than 0, make the current task ready again after `timeout` time ticks (wait forever if `timeout` is 0), set `*err` to `OS_TIMEOUT` and return. Otherwise, set `*err` to `OS_NO_ERR` and return.

- `UBYTE OSSemPost(OS_EVENT *pevent)`

  If there are tasks pending on semaphore `*pevent`, switch to the highest-priority task and (when the current task is resumed) return `OS_NO_ERR`. Otherwise, increment the semaphore counter and return `OS_NO_ERR`.

- `void OSStart(void)`

  Start actual multitasking: enable the timer used for task preemption and switch to the highest-priority ready task. This function is typically called after the user has created all necessary tasks, semaphores, etc.

- `UBYTE OSTaskCreate(void (OS_FAR *task)(void *pd), void *pdata, void *pstk, UBYTE p)`

  Create new task, with task body `pd`, initial argument `pdata`, preallocated working stack `pstk` and priority `p`. (The priority can range from 1 to 62; a lower number means a higher priority.) Return `OS_PRIO_EXIST` if there already exists a task with the same priority, or `OS_NO_ERR` otherwise.

- `void OSTimeDly(UWORD ticks)`

  Suspend the current task for `ticks` time ticks.

B   Porting Details

In this appendix, we show exactly where and how we modify uC/OS to operate on the X32. More specifically, we explain how we implement critical sections, context switching and preemption, and how we modify `OSIntExit` to check if there are other interrupts waiting to be serviced (Section 3.2).

Critical Sections   A "critical section" is a part of a program that can by no means be interrupted. To ensure uC/OS does not corrupt itself (e.g. starting a context switching during a context switch), most of its code is defined inside critical sections.

uC/OS defines two macros, to disable and enable interrupts at the start and end of critical sections, respectively. The X32 has no `STI`/`CLI` (set interrupt flag, clear interrupt flag) instruction pair, but uses a "global" interrupt to enable/disable all other interrupts. We use it to fill in the mentioned macros, as follows:

```
#define OS_ENTER_CRITICAL() \
    { DISABLE_INTERRUPT(INTERRUPT_GLOBAL); }
#define OS_EXIT_CRITICAL() \
    { if(OSRunning) ENABLE_INTERRUPT(INTERRUPT_GLOBAL); }
```

Context Switching Although the described context switch code has been made obsolete by the addition of similar functions to the X32 library, the working remains the same. The following code fragments reflect our switch to the new functions.

To initialize a task's working stack, we call `init_stack`, as follows (`ucos.c::OSTCBInit`):

```
ptcb->state[0] = (int)init_stack((void**)stk, (void*)task, (void*)pdata);
```

The arguments are a pointer to preallocated space for the working stack, the address of the respective task body and the task's initial argument. The address of top of the initially created stack is stored in the task's "task control block", and can be used to start the task later on.

To switch between tasks, we call `context_switch`, passing the address of the top of the target task's stack (`osspec.c::OSCtxSw`). To ensure the critical section around the call is maintained correctly, we set the X32 interrupt level to 1000 beforehand (this means no interrupt with priority under 1000 is serviced, i.e., all non-critical interrupts) and enable interrupts globally. When `RET` is executed to make the final switch to the target task, the task's interrupt level is restored; because the global interrupt flag is still on, normal interrupts are now serviced again.

```
set_execution_level(1000);
ENABLE_INTERRUPT(INTERRUPT_GLOBAL);
context_switch((void**)(OSTCBCur->state[0]), (void***)&(prev->state[0]));
```

Preemption To enable preemption, we use a specific X32 timer interrupt, with a priority of 50. The ISR calls the uC/OS scheduler to see if a task with a higher priority than the current task has become ready at this point, and if so, switch to it (`ucos.c::OSStart`).

```
isr_alarm() {
    OSTimeTick();
    OSSched();
}

peripherals[PERIPHERAL_TIMER1_PERIOD] = 20 * CLOCKS_PER_MS;

SET_INTERRUPT_VECTOR(INTERRUPT_TIMER1, isr_alarm);
SET_INTERRUPT_PRIORITY(INTERRUPT_TIMER1, 50);
ENABLE_INTERRUPT(INTERRUPT_TIMER1);
```

Interrupt Nesting As explained in Section 3.2, an interrupt may be preempted by another interrupt before it has had a chance to call `OSIntEnter`. This means the interrupt nesting level that uC/OS maintains may become 0, while there is still an interrupt waiting to be serviced. To avoid scheduling, and finish waiting interrupts instead, we check the execution level on the stack, to see if the current ISR has interrupted another ISR (`ucos.c::OSIntExit`).

```
fp = (unsigned int *)_get_fp();
fp = (unsigned int *)(*(fp-1));
exec_level = *(fp-4);
if ((..  && exec_level == 0) {
    ..
```

The first line uses an X32 library call to get a pointer to the current stack frame (the stack frame of the call to `OSIntExit`). The second line loads a pointer from this stack frame, to the stack frame above it (the stack frame of the call to the ISR). The third line loads the saved execution

level from this stack frame (the execution level of the code interrupted by the ISR). On the fourth line, we make sure only to schedule if the execution level is 0, i.e., the ISR did not interrupt another ISR.