PicOS

internals

Version 3.3
August, 2010

Table of contents

Preamble

This document describes the internal data structures, algorithms, and pieces of code used by the kernel. It may never be complete and is guaranteed to be occasionally outdated, but it may be useful in an emergency (e.g., if I am run over by a bus or something). I promise to update/expand it every once in a while.

1 Activities

We can talk of four types of activities, in the sense that any piece of code ever executed in the system runs within the framework of one of those types. They are:

- System initialization sequence
- Interrupt services
- The scheduler
- Processes

The system initialization code is only executed at reset (and power up). Its role is to properly initialize the relevant hardware registers (to select clock sources, preset I/O ports, pre-initialize devices and drivers) and set up the root process of the praxis. Note that some drivers may spawn their own (internal) processes. Also note that some driver-lookalike modules (notably PHY modules) are not initialized automatically, but require explicit actions (VNETI functions) invoked explicitly from the praxis to start them up.

At the end of the initialization sequence, the system creates the root process, assumes its identity, and issues a **`delay`** request for 16 milliseconds[1] with the target state of 0. This means that the formal startup of the root process (its first run perceived by the praxis) is delayed by 16 milliseconds to provide room for any internal processes (if present) to go first. The initialization sequence ends by entering the *scheduler loop*.

The unique power of PicOS, i.e., its powerful dynamics implemented within the trivially tiny resource base, combined with its amazing resilience to overloads and even occasional sloppiness in programming, results from the interplay of processes with interrupt service routines, which is coordinated by the scheduler. The underlying premise is the observation that while you cannot have everything (like multi-level interrupts and arbitrarily preemptible multiple tasks) within 2 KB of RAM, it is quite possible, with the proper selection of sacrifices, to create the kind of dynamics and responsiveness that will appear natural and convenient within the context of our class of applications, while being also extremely frugal in terms of memory and CPU requirements.

The functional perspective of processes in PicOS is described in the *PicOS* document. All processes share a single stack, which is also used by interrupts. By default, interrupts are limited to a single level. This means that at most one interrupt can be active (stacked) at any given moment. This is perfectly OK as long as the interrupt service routines are short and fast, which they always are. In contrast to other (more or less broken) systems (like TinyOS), whose interrupt service routines try to compensate for the lack of threads, PicOS can afford to make its interrupt service simple and non-preemptible. This is because the framework of its FSM-like threads is quite adequate for expressing the kind of practical dynamics needed for a convenient expression of reactive multi-threaded programs. In consequence, PicOS doesn't suffer from the stack runaway problem (not a single incident has ever happened in the whole history of the system) while offering such features as dynamic memory allocation, multi-threaded (safely

[1] Unless stated otherwise, whenever we say "millisecond", we have in mind PicOS's millisecond, i.e., 1/1024 of a second.

interleaved) activities, very good modularity, and even excellent accommodation of reasonable real-time requirements.

For those rare cases when a fast high-priority interrupt is truly required, PicOS allows selected interrupt service routines to yield to other interrupt service routines (the **NESTED_INTERRUPTS** option). That feature was sometimes used in the past to facilitate drivers for old RF chips that required precise program-driven signal shapers.[2] However, with the proper use of hardware tools (timers, compare and capture registers, etc.) such shapers can be implemented reliably while allowing for a reasonable lag in servicing the requisite interrupts. Consequently, there has been no demand for this option recently, even in the hard-real-time-conditioned praxes, like the BCG heart datalogger.

In a nutshell, the system works this way:

- There is a pool of processes. Each process can be *ready* (meaning it can use the CPU) or *waiting* (meaning it is waiting for some event). The processes are ordered rigidly in a somewhat accidental fashion, but mostly according to the time of their creation (see section 1.2).

- Whenever there is an event in the system that may result in a status change of some process from waiting to ready, the CPU scheduler is run. The scheduler scans the list of processes in the fixed order looking for the first process whose status says ready. Having found such a process, the scheduler runs it – by invoking the process function (sometimes referred to as the FSM function).

- Upon return from a process, when the process executes **release** (explicit or implicit), the scheduler scans the process queue from the beginning. This is because running a process can make other processes ready.

- If the CPU scheduler has examined all processes and found them all waiting, it halts the CPU. The CPU will be un-halted when an interrupt service routine decides that the scheduler should be run, i.e., there has been a process-waking event.

The conceptual role of interrupt service routines is to transform hardware events into logical events that wake up processes.

1.1 Processes and events

A process is described by a data structure called PCB (for Process Control Block). Here is its layout (see **PicOS/sysio.h**):

```
typedef struct    {
      word        Status;
      word        Timer;      /* Timer wakeup tick */
      fsmcode     code;       /* FSM function pointer */
      word        data;       /* Data pointer */
      event_t     Events [MAX_EVENTS_PER_TASK];
} pcb_t;
```

Attributes **code** and **data** point to the process's *code* and *data*, what else? Type **fsmcode** is declared (in **sysio.h**) as:

```
typedef void (*fsmcode)(word);
```

[2] In those cases, the option would allow UART interrupts to be preempted by the interrupts of the signal shaper/decoder.

and represents a pointer to the C function containing the process code method. For those processes that use private data (the so-called strands), the **data** pointer points to the respective structure or variable.

The **Status** word indicates whether the process is waiting for an event and, if this is not the case, specifies the state in which the process should be awakened when it receives the CPU. This is how its bits are interpreted:

If the TW bit is set, it means that the process is waiting for a timer (it has executed **delay**). In that case, the State field contains the target state number that was specified as the second argument of **delay**. The NE field specifies the total number of other (non-timer) events that the process is waiting for. Note that the (absolute) maximum on the number of such events is 7. This is also the maximum legitimate value of the compilation constant **MAX_EVENTS_PER_TASK** whose default setting is 4.

The way to check whether a process is ready or not is to look at the least significant 4-bit nibble of the **Status** word (see macro **waiting** in **kernel.h**). If that nibble is zero, it means that the process is ready and then the State field of the word indicates the process state to be assumed. The operation of waking up a process on a non-timer event puts the right value into the State field. For a timer event (**delay**) State already points to the right target state.

If the TW bit is set, the word **Timer** in the PCB describes the target delay (in the way that will be described later). Other events awaited by the process, if any, are represented in the **Events** array whose single element looks like this:

```
typedef struct    {
      word  State;
      word  Event;
} event_t;
```

where the **Event** word describes an event (a number which is often directly derived from some address), and **State** encodes the target state in this way:

Whenever a process issues a **when** request, the system takes the first free entry from the **Events** array in the process's PCB (based on the contents of NE in the Status word), fills it with the parameters of the request (event number, state), and increments NE.

The pool of processes known to the system is described in an array of PCBs (we call it the PCBT) whose (fixed) size is a compilation constant (it determines the maximum number of processes that the system can run simultaneously). This array is declared (in **kernel.c**) as:

```
__pi_pcb_t __PCB [MAX_TASKS];
```


Note that the PCB structure has no links that would assist in organizing the processes into direct queues. Those entries in **__PCB** that have **code == NULL** are deemed unused, and they are simply skipped by the system when searching for a particular process. In particular, when looking for a process to run, the CPU scheduler scans the entire array. The little overhead on skipping the unused entries is compensated by the lack of overhead on maintaining links.

Process are identified (internally and also by the praxis) by the addresses of their PCBs. In particular, the value returned by **runfsm** (which translates into a call to the system function **__pi_fork**) is the PCB address of the created process cast to type **sint**. When a process is run, the system variable **__pi__curr** (current) points to its PCB. Many operations refer to the current process by default.

The standard way to look up a process or to do something for all processes involves a loop through the PCBT. Here is a macro (defined in **kernel.h**) to start such a loop:

```
#define FIRST_PCB         (&(__PCB [0]))
#define LAST_PCB          (FIRST_PCB + MAX_TASKS)
#define for_all_tasks(i)  for (i = FIRST_PCB; \
                               i != LAST_PCB; i++)
```

and here is a list of simple macros for interpreting and modifying the **Status** word (assuming that the argument points to the PCB of the process being attended to:

```
#define incwait(p)      ((p)->Status++)
#define inctimer(p)     ((p)->Status |= 0x8)
#define waiting(p)      ((p)->Status & 0xf)
#define twaiting(p)     ((p)->Status & 0x8)
#define nevents(p)      ((p)->Status & 0x7)
#define tstate(p)       ((p)->Status >> 4)
#define settstate(p,t)  ((p)->Status = ((p)->Status & 0x7) \
                                     | ((t) << 4))
#define prcdstate(p,t)  ((p)->Status = (t) << 4)

#define wakeupev(p,j)   ((p)->Status = \
                            (p)->Events [j].State)
#define wakeuptm(p)     (p)->Status &= 0xfff0
#define cltmwait(p)     (p)->Status &= 0xfff7
```

Here is a macro for setting up a new wait request (event) entry in the PCB:

```
#define setestate(e,s,v)  do { \
                              (e).State = (s) << 4; \
                              (e).Event = (v); \
                          } while (0)
```

Macro **wakeupev** from the first list shows how a process is awakened by a non-timer event: the State field from the respective **Events** entry (number **j**) is copied to the State field of the **Status** word, and the least significant nibble of the **Status** word is cleared. The case of waking up a process from the timer (**delay**) is even simpler. As the setting of State in **Status** is already correct, all that remains to be done is to zero out the rightmost nibble of **Status**.

For illustration, let us see the complete code of **when** (named **__pi_wait** in **kernel.c**):

```
void __pi_wait (word event, word state) {
      int j = nevents (__pi_curr);

      if (j == MAX_EVENTS_PER_TASK)
            syserror (ENEVENTS, "sw");
      setestate (__pi_curr->Events [j], state, event);
      incwait (__pi_curr);
}
```

The function starts by loading the index of the first free entry in **Events** into **j** (which is the same as the number of events already awaited by the process). Then it checks whether the process hasn't already reached the maximum. Next, the function sets the new entry in **Events** and increments the number of awaited events in **Status**.

Here is the code of operation **trigger** which delivers a **when** event:

```
void __pi_trigger (word event) {
      int j;
      __pi_pcb_t *i;

      for_all_tasks (i) {
            if (i->code == NULL)
                  continue;
            for (j = 0; j < nevents (i); j++) {
                  if (i->Events [j] . Event == event) {
                        wakeupev (i, j);
                        break;
                  }
            }
      }
}
```

The function scans through the PCBT, and for every nonempty slot examines its list of awaited events looking for one matching the argument. If a matching entry is found, the process is awakened.

1.2 Scheduling

The scheduling algorithm is extremely simple and naive. Despite numerous temptations (and some actual attempts) to fix it, there has been absolutely no need so far to spoil its simple beauty.

As described in the previous section, processes occupy slots in the PCBT, which is a fixed-size array of PCBs. When a new process is run (forked), the system will put its PCB into the first free slot looking from the beginning (as implied by **for_all_tasks** – see page 6). When a process terminates, its slot is vacated. Consequently, as processes come and go, their allocation to slots may be somewhat accidental with a strong tendency to leave unoccupied slots at the end of the PCB array.

Nonetheless, the allocation of PCBs determines process priorities. If two processes are ready, the one whose PCB is closer to the front of the array will be given the CPU first. For as long as the processes are not CPU bound (and in most cases they are not), it

mostly doesn't matter. If you think it does matter, then you may try to create them in the proper order – to make sure that the more important ones are created first.

I have thought of a few simple tricks that would help in such circumstances (i.e., when the order of processes in the PCB array does matter). For example, there could be a variant of **fork** allocating the PCB from the end (say, to create a low-priority process). Or perhaps, PCBs should be allocated from the end by default, except when explicitly creating a high priority process. Finally, there could be pointers (which I would much prefer to avoid) organizing the processes into flexible prioritized lists. The options are there, so it is easy to fix the problem, except that there doesn't seem to be any as of now.

Let us now have a look at the CPU scheduler loop (file **kernel/scheduler.h**):

```
{
	SET_RELEASE_POINT;
Redo:
	update_n_wake (MAX_UINT);
	for_all_tasks (__pi_curr) {
	    if (__pi_curr->code != NULL && !waiting (__pi_curr)) {
		(__pi_curr->code) (tstate (__pi_curr));
		goto Redo;
	    }
	}
	SLEEP;
	goto Redo;
}
```

The real version is a bit more messy, because of some conditionally compiled code being mostly a legacy of exotic requirements for some "special" praxes (e.g., the notorious GENESIS). The above chunk is inserted into the (architecture-specific) initialization code, at the very end. Note that the state in which the process is to be activated is passed as the argument of the FSM function. The data pointer is available to the process as **__pi_curr->data** (which can be referenced via a suitable alias).

Here is a sanitized version of the "main program" for MSP430 (system execution starts from there):

```
	int main (void) {

		ssm_init ();	// Clock, ports, and other basics
		mem_init ();	// Memory
		ios_init ();	// Drivers
		tcv_init ();	// VNETI

		// Assume root process identity
		__pi_curr = (__pi_pcb_t*) fork (root, NULL);
		// Delay root startup for 16 msec to give leeway
		// to driver processes
		delay (16, 0);
		// Power-up mode by default
		powerup ();
		// Enable the interrupt system
		_EINT ();
		// Fall through to scheduler loop
	#include "scheduler.h"
```


```
}
```

By setting **__pi_curr** to the "process ID" of the newly created **root** process, the system makes sure that the subsequent operations will be executed as if they were issued from that process's code method. Thus, the root process is told to continue at state 0 after 16 milliseconds. The scheduler loop is organized in such a way that its very beginning (at label **Redo**) is the point to be branched to after a process performs **release**. This return point is also marked with a callable address (looking like a function that can be invoked by a process) with **SET_RELEASE_POINT**. Here is the MSP430 version of this macro together with the requisites (see file **MSP430/arch.h**):

```
#define SET_RELEASE_POINT __asm__ __volatile__ (\
      ".global __pi_release\n"\
      "__pi_release: mov %0, r1"\
            :: "i"(STACK_START): "r1")

void __pi_release () __attribute__ ((noreturn));
#define release __pi_release ()
```

SET_RELEASE_POINT declares a global entry point (named **__pi_release**) which amounts to something that can be formally called as an argument-less function. When called, that "function" will never return: it resets the stack pointer (register **r1** on MSP430) to the initial value (corresponding to "nothing on the stack") and falls through to the scheduler loop. Operation **release** is implemented as a call to that dummy function. This way it can be performed from a nested function (which at the bottom has been invoked from a process code method) and it will always move control completely out of the process to level-zero stack at the beginning of the scheduler loop.

The proper scheduler loop starts at label **Redo** with a call to **update_n_wake**. This function is responsible for updating the system clock (used for measuring **delay** time) and waking up those processes whose **delay** timers have expired. Its role is described in detail in section 1.3.

The rest of the scheduler loop is very straightforward: it locates the first ready process (the PCB entry is not free and the process is not waiting for anything) and calls its FSM function passing it the state number as the argument. Whenever that happens (i.e., a ready process is found and its function is called and it subsequently returns), the scheduler starts from the beginning.[3] If no ready process is available, the scheduler executes **SLEEP**, which is a macro with the following expansion (in its MSP430 version, see **MSP430/mach.h**):

```
#define SLEEP do { \
            if (__pi_systat.pdmode) { \
                  cli; \
                  if (__pi_systat.evntpn) { \
                        sti; \
                  } else { \
                        _BIS_SR (LPM3_bits + GIE); \
                  } \
            } else { \
                  cli; \
```

[3] Note that when the FSM function executes **return**, the scheduler loop will be automatically entered from the beginning (as after **release**). A simple **return** (or fall through the end of function) from the code method is just one (and not very popular) way of releasing the CPU.

```
                if (__pi_systat.evntpn) { \
                      sti; \
                } else { \
                      _BIS_SR (LPM0_bits + GIE); \
                } \
          } \
          __pi_systat.evntpn = 0; \
    } while (0)
```

The two parts of the **if** statement are identical except for using **LPM3_bits** versus **LPM0_bits** in the **_BIS_SR** statement, depending on the value of **__pi_systat.pdmode**. That is a bit flag indicating whether the system is operating in the *power-down* mode (in which case the bit is set), or in the "normal" (full power) mode, if it isn't. A good understanding of the **SLEEP** sequence is key to the grasp of practically all synchronization problems in PicOS, so let us analyze this sequence in detail.

The global variable **__pi_systat** is a **word**-sized collection of bits with the following layout (MSP430 version, file **MSP430/arch.h**):

```
typedef struct {
      byte  pdmode:1,   // Power down flag
            evntpn:1,   // Scheduler event pending
            fstblk:1,   // Fast blink flag
            ledblk:1,   // Blink flag
            ledsts:4;   // Blink status of four leds
      byte  ledblc;     // Blink counter
} systat_t;
...
volatile systat_t __pi_systat;
```

The first nibble of the first byte provides four bit flags of which the first two appear in the **SLEEP** sequence. The remaining flags, and in fact all the remaining fields in **__pi_systat**, are used to implement blinking LEDs (we can leave them until later).

The sole purpose of **pdmode** is to select the bit pattern to be written into the CPU status register (SR) when the CPU is put to sleep. When **pdmode** is 0, which selects the normal operation, those bits are described by the constant **LPM0_bits**, which basically means shallow sleep (see the MSP430 manual for details) with a relatively large power consumption. **LPM3_bits**, on the other hand, selects the deepest possible sleep from which the CPU is still able to wake up on an event (through an interrupt). The tradeoff between these modes involves a slightly longer wakeup time from the deep sleep mode, which means that the praxis should generally know what it is doing when it decides in which mode to run. PicOS provides operations (**powerdown ()** and **powerup ()**) for changing the sleep mode (which effectively amount to modifying the **pdmode** flag) from the praxis.

Without loss of generality, we can consider only one of the two segments of the **if** statement in the **SLEEP** sequence, e.g., **pdmode == 0**. What happens is this:

1. All interrupts are temporarily masked out. For MSP430, the **cli** macro expands as **_DINT()**. Also, **sti** expands into **_EINT()**.

2. It is checked whether the **eventpn** flag from **__pi_systat** is on. If this happens to be the case, interrupts are unmasked and the **SLEEP** operation has no effect. Note that **eventpn** is reset before the operation exits.

3. If **eventpn** is zero, the CPU is put to sleep (according to the mode in effect) with interrupts enabled (the GIE flag in SR is set).

In the latter case (i.e., the CPU has been put to sleep), the only way for the scheduler to receive control is when an interrupt service routine decides that it makes sense to run the scheduler loop. That means that the interrupt has (or may have) rendered some process(es) ready. Note that the reason why the sleep state has been entered in the first place was that, at some point, no process was found ready to run. On MSP430, an interrupt service function concluding that the CPU should leave the sleep state has to erase the sleep bits in the copy of SR that was pushed on the stack when the interrupt was received. This way, when the function returns, the re-loaded SR will have those bits cleared and the **SLEEP** sequence will continue past the **_BIS_SR** statement. The exact operation required to accomplish this from an interrupt service function is:

```
_BIC_SR_IRQ (LPM4_bits)
```

executed before **return**. The argument (**LPM4_bits**) describes the full bit mask for all the possible sleep bits in SR (which are cleared by the instruction).

The role of **eventpn** is to make the actual operation a bit more structural and, more importantly, to account for a race condition between interrupts carrying events that may wake up processes, and those processes preparing to wait for those events. Imagine the scenario where a process (e.g., a device driver) wants to extract a byte of data from a device. The byte is not available immediately, so the process has to put itself to sleep awaiting an event that will be triggered when the data shows up. The code may look like this:

```
state GET_BYTE:
      if (!byte_available) {
            when (DRIVER_EVENT, GET_BYTE);
            release;
      }
      extract_byte;
      ...
```

First of all, interrupts being asynchronous with processes, it may happen that the byte will show up after **byte_available** was checked (and found 0) and before the **when** operation has been given a chance to complete. This way, the interrupt service routine will find no process waiting (so it wake up nobody) and the process will end up waiting for something that is already available. Consequently, the driver will try to make sure that the event cannot be presented within the critical sequence (when its occurrence can be overlooked), e.g.,

```
state GET_BYTE:
      mask_device_interrupts;
      if (!byte_available) {
            when (DRIVER_EVENT, GET_BYTE);
            unmask_device_interrupts;
            release;
      }
      extract_byte;
      unmask_device_interrupts;
      ...
```


Note that this scheme requires a certain discipline regarding how the wait (**when**) requests are issued and handled. For example, the driver process cannot issue other **when** (or **delay**) requests after the critical one, unless the device interrupts remain masked, as that would introduce another race. This is because formally the process is put to sleep when it executes **release**. When the event is delivered, the interrupt service routine will execute something similar to **__pi_trigger** (page 7). If following that operation the process keeps running in the same state issuing other wait requests, those requests will override the prematurely triggered device event. This observation can be translated into the following rule:

> A process issuing a wait request for an event that will be delivered by an interrupt service routine must make sure that interrupts from the device are disabled before the *availability condition* is checked and they remain disabled until the process has issued its last wait request in the present state before releasing the CPU.

For as long as this condition is fulfilled, the event will not be lost, in the sense that the interrupt service routine will correctly locate the event and mark the process as runnable. Note that this may happen before the process in fact executes **release** (interrupts are re-enabled before that), but this is OK. The process will appear ready for the next turn of the scheduler loop.

This however, hints at another race condition inherent in the scheduler. Suppose that the process has gone through the drill outlined above and it has executed **release** after unmasking device interrupts. The scheduler goes through another loop (which it always does after a **release**) and finds that all processes are now waiting. So it is just about to put the CPU to sleep. Now suppose that the interrupt from the device occurs a short while before that. What is going to happen? Even though the interrupt service routine will mark the process as ready, the scheduler has checked and already concluded that no process is ready, so it will put the CPU to sleep anyway awaiting a moment when an interrupt service routine removes the sleep state. The event hasn't been overlooked by the process (which took proper precautions by masking interrupts from the device), but it has been missed by the scheduler which has missed the fact that the process has become ready *after* the last check.

The primary role of **eventpn** (for event pending) is to resolve this race. The idea is that any interrupt service routine delivering a process waking event is supposed to set this flag. Just before actually putting the CPU to sleep, the scheduler masks (for a tiny while) *all* interrupts and looks at the flag. If the flag is found set, the scheduler clears it and goes through one more loop; otherwise, the CPU is in fact put to sleep. Note that the operation of entering the sleep state happens simultaneously (atomically) with re-enabling interrupts (the GIE flag), which makes sure that nothing can be lost (there is no more race).

An interrupt service routine that delivers a process-waking event will execute this operation (MSP430 version, file **MSP430/mach.h**):

```
#define RISE_N_SHINE __pi_systat.evntpn = 1
```

and this one upon return:

```
#define RTNI do { \
                  if (__pi_systat.evntpn) \
                        _BIC_SR_IRQ (LPM4_bits); \
                  return; \
              } while (0)
```


One additional advantage of having the flag is that **RISE_N_SHINE** (also known as a *scheduler kick*) can be executed from a (regular) function that has been called by an interrupt service routine, while **_BIC_SR_IRQ** only makes sense from the very interrupt service routine (when the stack is just one level above the scheduler loop).

Note that operations **when** and **trigger**, with the latter issued from a process, are free of the kind of races described above. This is because processes are run by the scheduler in a synchronous fashion (a process is only run when the scheduler explicitly invokes its code function). Interrupt service routines, on the other hand, can run at any time (unless the requisite interrupts are masked), which makes them susceptible to races with the scheduler (and thus processes).

1.3 System clocks

PicOS kernel uses three conceptually separate general purpose clocks. Specific device drivers may need other clocks, which they can implement using various hardware timers available in the CPU, except for those that are used by the standard clocks. The three standard clocks are:

1. The *seconds* clock. This clock is used to measure absolute time in seconds counted from the moment of last reset.

2. The *strobe* clock (a.k.a. the *auxiliary* clock). This clock provides millisecond-grained strobes for implementing utimers, blinking LEDs, low-level debouncers for I/O pins, etc.

3. The delay clock. This clock is used to implement the **delay** operation.

We confine our discussion to MSP430. The way the system clocks are organized is a bit messy, because of several options (and lots of conditional compilation) mostly resulting from the legacy requirements for some exotic boards (here goes again the abominable GENESIS). In a "normal" situation, when the primary crystal connected the CPU is the 32768 Hz low-power watch crystal (as it should always be), the picture is simple and clean. We shall discuss things assuming that this is the case and then digress a bit towards pathology.

The delay clock

This is the clock that is closest to the praxis, as typical processes make extensive use of the **delay** operation. This operation is implemented as follows.

The system maintains three **word** variables named **__pi_old**, **__pi_new**, and **__pi_mintk**. For aesthetic reasons, we shall strip the prefix **__pi_** from their names in the following discussion calling those variables OLD, NEW, and MINTK. Conceptually, OLD and NEW keep track of time in milliseconds modulo 64K (i.e., within the last 60 real seconds). However, for as long as there is no need to run the delay clock (i.e., no process is waiting on **delay**) the time is not (or need not be) advanced.

The reason why two variables are needed (instead of just one) is that we would prefer to avoid keeping the precise track of the passed number of milliseconds (which would require an interrupt every millisecond), except for those moments that are actually meaningful (like when the delay timer of a process actually expires). In a nutshell, the idea is this: OLD tells the millisecond time (modulo 64K) of the last moment when the system *handled* the delay clock, while NEW tells the most current setting of that clock as reported by the hardware timer. In other words, OLD is the backpointer of the timer: up

to that point in time things have been accounted for, while NEW is the forward pointer: this is how far the real timer has advanced. The discrepancy between the two pointers results from the fact that the delay timer is handled in the scheduler, which can incur a lag. One of the objectives of the handler is to make the two pointers meet.

One of the reasons why the actual progress, as far as processes waiting on **delay** are concerned, is made in the scheduler is that we want to keep the interrupts (and especially timer interrupts) as simple as possible (because they are not preemptible). The second reason is that we want to avoid the synchronization problems (meaning complications) that would ensue if interrupt service routines were directly responsible for waking up processes waiting on **delay**.

The requisite processing is done in function **update_n_wake**, which is called upon entrance to the scheduler loop (page 8). If the function finds that NEW is different from OLD, it will bring the two pointers together, i.e., update OLD to be equal NEW. It will also wake up any processes whose **delay** timers have expired. One important property of the scheme is that OLD can only be updated (modified) by **update_n_wake**, while NEW can only be modified by the hardware timer (an interrupt service function). This way the two sides of the mechanism operate on the opposite ends of the lag interval and avoid getting "in the way".

The third variable, MINTK, is set to the target millisecond count for the first (earliest) process waiting on the **delay** timer to be awakened. When **update_n_wake** is called it first checks whether MINTK falls between OLD and NEW (accounting for the wraparound modulo 64K). The exact condition (defined as a macro in **kernel/kernel.h**) is this:

```
#define twakecnd(o,n,m) ( ( (m) <= (n) && (m) >= (o) ) || \
      ( ((n) < (o)) && ( ((m) <= (n)) || ((m) >= (o)) ) ) )
```

Despite the cryptic look, it is conceptually quite simple. With **o**, **n**, **m** standing for OLD, NEW, and MINTK, respectively, we say that MINTK falls (timewise) between OLD and NEW, if either NEW >= OLD and OLD <= MINTK <= NEW, or NEW < OLD (we have a wraparound) and MINTK <= NEW or MINTK >= OLD. The macro encodes a somewhat optimized version of exactly this condition.

If the condition holds, it means that at least one process has to be awakened. The role of MINTK is to facilitate a quick check whether the pool of processes should be traversed looking for processes to wake up. For simplicity (and economy of space), there is no explicit queue that would link only those processes that are waiting for the timer (preferably in the increased order of their wakeup time), so once we decide to check, we have to scan them all. However, we only do so when there actually is a process to wake up, as determined by MINTK. Here is the code of **update_n_wake** (to be found in **kernel/kernel.c**):

```
void update_n_wake (word min) {
  pcb_t *i;
  word d;
  TCI_UPDATE_DELAY_TICKS;
  if (twakecnd (__pi_old, __pi_new, __pi_mintk)) {
    for_all_tasks (i) {
      if (!twaiting (i))
        continue;
      if (twakecnd (__pi_old, __pi_new, i->Timer)) {
        wakeuptm (i);
      } else {
```


```
            d = i->Timer - __pi_new;
            if (d < min)
              min = d;
          }
        }
      } else {
        if (__pi_mintk - __pi_new < min)
          goto MOK;
      }
      __pi_mintk = __pi_new + min;
MOK:
      __pi_old = __pi_new;
      TCI_RUN_DELAY_TIMER;
}
```

The function starts by executing **TCI_UPDATE_DELAY_TICKS**. This is a macro which expands into an architecture-specific operation to obtain the most current reading of NEW from the hardware timer. In some implementations, this operation may be void, e.g., when NEW is updated by interrupts occurring every millisecond. However, the recommended implementation of the delay timer postulates that timer interrupts be triggered as sparsely as possible, preferably only at those moments when a process's **delay** timer expires. Generally, hardware timers have a limited flexibility. For example, the standard implementation of the **delay** clock for MSP430 uses a timer that cannot tick slower than once per 16 seconds. As the maximum legitimate **delay** interval is 64K-1 milliseconds (i.e., almost 60 seconds), it may happen that some timer interrupts will be spurious in a sense, i.e., the advancement of NEW signaled by them will not result in a process wakeup.

Thus, the solution decouples the interpretation of time advancements from the actual events. It assumes that the hardware advances the **delay** clock (specifically NEW) in some fashion to make sure that the target events are never missed, but not necessarily in the precise discrete steps corresponding solely to those events. When a timer interrupt detects that MINTK has fallen between OLD and NEW, it will **RISE_N_SHINE** the scheduler to run the loop and wake up the respective process(es). Otherwise, it will just update NEW. This does not preclude the scheduler loop being executed for other reasons. Also, running processes and going through multiple turns of the loop may result in a lag whereby the last report of NEW becomes outdated, while the interrupt (to convey the next eventful update may be still far ahead).

Consequently, whenever **update_n_wake** is executed it will poll the timer (via **TCI_UPDATE_DELAY_TICKS**) for the most recent reading (i.e., the most up to date value) of NEW. As we shall see, **update_n_wake** is also executed when a process issues a **delay** request – to make sure that the reference point in time (against which the new **delay** timer is being set) is up to date regardless of any lags or delays in reporting (via interrupts) only those updates of NEW that are considered eventful from the viewpoint of the processes that are already waiting on the timer.

Having made sure that NEW is up to date, the function evaluates the wakeup condition for the first-to-be-awakened process (whose target millisecond tick is stored in MINTK). If the test succeeds, the function runs through the processes, identifies those waiting on the timer (the requisite macros were shown on page 6) and checks whether they should be awakened. If a process still has to wait, its target wakeup time is used to calculate the new value of MINTK.

The argument to the function specifies the starting value for the calculation of the new minimum delay, which will be transformed into the new setting of MINTK. When the function is called from the scheduler, that starting value is **MAX_UINT**, so it doesn't matter. As we shall see, when the function is called from **delay**, its argument corresponds to the amount of delay requested by the new process (which should take part in the calculation of new MINTK).

If no process is to be awakened, the old MINTK value is retained unless the function's argument specifies a lower delay (counting from now), in which case it prevails. At the end, the function sets OLD = NEW and executes **TCI_RUN_DELAY_TIMER**. The role of that architecture-specific operation is to set the hardware alarm clock (the interrupt) to occur at least at the time specified by the new value of MINTK. Note that the alarm clock is set even if no process is waiting on a **delay** timer; in such a case, it will be set for the maximum possible interval. This is simpler and less expensive that imposing more subtle conditions that would inflate the code. For MSP430, it means one spurious interrupt after 16 seconds. Under normal circumstances, some processes will most likely issue delay requests in the meantime resetting the alarm clock before it generates that idle tick, which will do no harm anyway.

Note that the fact that MINTK fulfills the condition for wakeup need not imply that a process will be awakened (so the process lookup loop may be occasionally entered unnecessarily). For one thing, MINTK is set to something even if no process is waiting on the timer. It may also happen that the process for which MINTK was last set has been awakened by another event and is no more waiting for the timer. All such cases are perfectly OK and their impact on the scheme's performance is statistically insignificant.

Here is the simple code of **delay**:

```
void delay (word d, word state) {
      settstate (__pi_curr, state);
      cltmwait (__pi_curr);
      update_n_wake (d);
      __pi_curr->Timer = __pi_old + d;
      inctimer (__pi_curr);
}
```

See page 6 for the requisite macros. The function starts by setting the State field of the process's **Status** to the target state of the **delay** request. Then it removes any present **delay** flag from **Status**. This is because the process may have issued a **delay** request before (in its present state) which the new request will override. That previous request is removed before calling **update_n_wake**, because we don't want it to be considered as a legitimate member of the current **delay** pool. The role of **update_n_wake** is twofold: first, it makes sure that the reference time for the new request is up to date; second, it recalculates MINTK taking into account the new **delay** value of the current process. Finally, the **Timer** attribute of the PCB is set to the target tick and the process is marked as waiting for the timer.

The strobe clock

This clock has three built-in applications at present while being open for praxis-specific (or board-specific) applications. The architecture-independent components assume that the strobe clock is implemented as an interrupt service routine that is run every millisecond for as long as *any* of the components indicates that it still needs the clock. if the need disappears, the architecture-specific part is free to halt the interrupts. When a

component becomes active again, it will communicate that fact to the system, such that the timer can be resumed.

The three built-in components are:

- Blinking LEDs
- Debouncer for special I/O pins, a.k.a. COUNTER and NOTIFIER
- Utimers

Let us have a look at the case of blinking LEDs which best illustrates the concept. File **irq_timer.h** in directory **PicOS** contains conditional inclusion of the specific component files (based on the setting of the respective configuration constants). Note that the utimers component is not mentioned there because it is not optional. On the other hand, the LEDs component (file **irq_timer_leds.h**) is only included if **LEDS_BLINKING == 1**.

The component files are concatenated together and inserted as a single chunk of code into the interrupt service routine of the strobe timer. This somewhat clumsy way of "modularizing" the components results from the need to maintain eCOG1-compatibility of the system. With Cyan SDK for the eCOG1, interrupt service routines cannot call functions, unless those functions are declared in a cumbersome and messy way. Consequently, it is easier to insert a chunk of code directly into the code of an interrupt service routine than to provide a function callable from that routine. Here is the chunk that gets included as the blinking LEDs module:

```
if (__pi_systat.ledsts) {
      if (__pi_systat.ledblc++ == 0) {
            if (__pi_systat.ledblk) {
                  if (__pi_systat.ledsts & 0x1)
                        LED0_ON;
                  if (__pi_systat.ledsts & 0x2)
                        LED1_ON;
                  if (__pi_systat.ledsts & 0x4)
                        LED2_ON;
                  if (__pi_systat.ledsts & 0x8)
                        LED3_ON;
                  __pi_systat.ledblk = 0;
            } else {
                  if (__pi_systat.ledsts & 0x1)
                        LED0_OFF;
                  if (__pi_systat.ledsts & 0x2)
                        LED1_OFF;
                  if (__pi_systat.ledsts & 0x4)
                        LED2_OFF;
                  if (__pi_systat.ledsts & 0x8)
                        LED3_OFF;
                  __pi_systat.ledblk = 1;
            }
            if (__pi_systat.fstblk)
                  __pi_systat.ledblc = 200;
      }
      TCI_MARK_AUXILIARY_TIMER_ACTIVE;
}
```

The **ledsts** attribute of **__pi_systat** (see page 10) is a 4-bit nibble with one bit for each of the up to four LEDs handled by the standard LEDs driver. When a given bit is 1,

it means that the corresponding LED is supposed to blink. Thus, the first **if** condition determines if any of the LEDs is blinking. If not, then the entire chunk of code is bypassed.

Otherwise, **ledblc** is used as a counter (note that it is a whole byte). The counter is incremented by one and when it spills into 0, the blinking LEDs are turned on or off, depending on the current value of **ledblk** (a bit flag), which is subsequently flipped. Next, if **fstblk** is set (meaning fast blinking), **ledblc** is initialized to 200; otherwise, it will start from zero (where it has just ended) which will give it a much higher interval to the next spill.

At the end, for as long as any of the LEDs is still blinking, the chunk executes **TCI_MARK_AUXILIARY_TIMER_ACTIVE**, which is a declaration that the timer is still needed, i.e., the next interrupt is expected one millisecond from now. If the interrupt service routine runs all the chunks registered as components of the strobe clock and finds that none of them has executed **TCI_MARK_AUXILIARY_TIMER_ACTIVE**, it will conclude that the timer is not needed any more. The macro is in principle architecture-dependent (basically it sets a Boolean flag – see page 20).

The seconds clock

The role of this clock is to count seconds since system reset. The accumulated number of seconds is returned by the **seconds** function. This clock can be implemented by a dedicated timer interrupt occurring every second and adding 1 to the global variable **__pi_nseconds** of type **lword**.

File **second.h** in directory **PicOS** includes optional chunks of code that can be inserted into the function of the seconds clock in a manner similar to the strobe clock. All those chunks are concatenated and run once every second immediately *after* **__pi_nseconds** has been incremented by 1. File **board_second.h** from the board directory (which is one of the mandatory board definition files) is automatically included as the first chunk.

Another standard file included from **second.h** is **second_reset.h** (in directory **PicOS**) which contains conditional code for resetting the board on a predicate that can be declared (as a macro) by the praxis. Such a predicate will be evaluated every second. Here are the contents of **second_reset.h** which are self-explanatory:

```
#ifdef EMERGENCY_RESET_CONDITION

if (EMERGENCY_RESET_CONDITION) {
      watchdog_stop ();
      cli;

#ifdef EMERGENCY_RESET_ACTION
      EMERGENCY_RESET_ACTION;
#endif
      reset ();
}
#endif
```

Several boards use this mechanism to implement a soft reset button. For example, in **MSP430/BOARDS/WARSAW** we see these definitions (see files **board_pins.h** and **board_second.h**):

```
#define SOFT_RESET_BUTTON_PRESSED ((P2IN & 0x10) == 0)
```

and

```
#define EMERGENCY_RESET_CONDITION SOFT_RESET_BUTTON_PRESSED
...
```

The reset condition describes the low state of pin P2.4 which is connected to a soft reset button. When pressed, the button grounds the pin, which is otherwise pulled up by a resistor.

Low level implementation of clocks

In this section we will have a look at one implementation of the low-level events (interrupts) needed to run the three system clocks on MSP430. There are two such implementations selected with compilation constants. We shall discuss the default and preferred one (used on boards with low-speed primary crystal) a.k.a. the **`TRIPLE_CLOCK`** option.

The implementation uses one hardware timer and three of the set of its associated CCRs (capture/compare registers). Timer B is used on those CPUs on which it is available; otherwise, e.g., on CC430, the first Timer A (number 0) is used.

The timer is set to operate in the so-called *continuous* mode, whereby its counter register is incremented continuously from zero to 64K−1 and then again from zero. The increment rate is set at 1/8 of the ACLK frequency which, assuming that ACLK is driven by a 32768 Hz crystal, yields 4096 increments per second.

Each of the three clocks uses a separate CCR: CCR0 is used by the delay clock, CCR1 by the seconds clock, and CCR2 by the strober. A CCR will generate an interrupt when the counter reaches its current setting. Then, if a next event is required, the interrupt service routine must explicitly set (increment) the CCR to a new value, according to the required interval until the next interrupt.

CCR1 and CCR2 share the same interrupt vector, while CCR0 uses a separate vector of its own.[4] Here is the interrupt service routine that handles both CCR1 and CCR2:

```
interrupt (TCI_VECTOR_S) timer_auxiliary () {

      word aux_timer_inactive;

      // This test also removes the interrupt status
      if (TCI_AUXILIARY_TIMER_INTERRUPT) {
            TCI_CCA += TCI_HIGH_DIV;
            aux_timer_inactive = 1;
            // Take care of utimers
            if (__pi_utims [0] == 0)
                  goto EUT;
            if (*(__pi_utims [0])) {
                  (*(__pi_utims [0]))--;
                  aux_timer_inactive = 0;
            }
            ...
EUT:
```

[4] This is a property of MSP430 hardware. All CCR registers except CCR0 share one interrupt vector, while CCR0 has a separate interrupt vector just for itself.

```
#include "irq_timer.h"

            if (aux_timer_inactive)
                  // Nobody wants us any more
                  cli_aux;
            RTNI;
      }

      // The seconds clock
      TCI_CCS += TCI_SEC_DIV;
      __pi_nseconds++;

      check_stack_overflow;

#include "second.h"

      RTNI;
}
```

Note that the names of the timer registers referenced in the above function are covered with PicOS's private macros (because different actual registers may be used depending on the CPU model). Those macros are defined in **MSP430/mach.h**. For example, here is the set of definitions specific to Timer B:

```
#define TCI_CCR   TBCCR0      /* delay */
#define TCI_CCS   TBCCR1      /* seconds */
#define TCI_CCA   TBCCR2      /* strober */
#define TCI_VAL   TBR         /* counter register */
#define TCI_CTL   TBCTL       /* control register */
#define TCI_VECTOR   TIMERB0_VECTOR /* vector 1: CCR0 */
#define TCI_VECTOR_S TIMERB1_VECTOR /* vector 2: CCR1+2 */
// Tells CCR2 from CCR1
#define TCI_AUXILIARY_TIMER_INTERRUPT (TBIV == 4)
#define sti_aux   _BIS (TBCCTL2, CCIE) /* strober enable */
#define cli_aux   _BIC (TBCCTL2, CCIE) /* .. and disable */
#define sti_sec   _BIS (TBCCTL1, CCIE) /* seconds enable */
#define cli_sec   _BIC (TBCCTL1, CCIE)
#define sti_tim   _BIS (TBCCTL0, CCIE) /* delay enable */
#define cli_tim   _BIC (TBCCTL0, CCIE)
```

Operations **sti_*xxx*** and **cli_xxx** respectively enable and disable interrupts for the corresponding CCR, which has the effect of logically switching the clock on and off. The counter register (**TCI_VAL**) is shared by the three clocks and it never stops running. Normally, it is driven by the low-speed crystal (ACLK) whose operation is retained in the deepest (recoverable) sleep mode of the CPU and costs the minimum amount of power.

The **if** condition determines whether the interrupt being handled has been triggered by CCR1 or CCR2. The requisite macro (**TCI_AUXILIARY_TIMER_INTERRUPT**) looks at the value in the timer's IV (interrupt vector) register, which has the additional effect of clearing the interrupt condition. If we are handling a strober event (the part within the **if**), we begin by incrementing the respective CCR (**TCI_CCA**) by **TCI_HIGH_DIV**, which is a symbolic name for the constant 4 (this many ticks of the counter amount to 1 millisecond). This way the CCR becomes immediately set to trigger another interrupt exactly one millisecond later. Whether this interrupt will in fact occur depends on the setting of **aux_timer_inactive** at the end of the sequence of instructions that the function executes next. This flag is used to tell the function, after it has run through its

chain of actions, whether any of those actions expects the clock to be active for the next turn.

Having initialized **aux_timer_inactive** to 1, the function first looks at the utimers. There are four slots (addresses) for utimers and the code examining them is unwound for speed (there is no need to look at all four identical cases). A utimer slot is either occupied, in which case it contains the address of a word containing a value to be decremented towards zero, or is empty, in which case it contains 0 (NULL). The first empty slot terminates the scanning (utimers are stored without holes).

If an non-empty utimer slot points to a word containing nonzero, that word is decremented and **aux_timer_inactive** is cleared. Otherwise, the utimer has run down to its target of zero and needs no more clock events (until reset by its owner), so the flag is left intact. Note that technically **aux_timer_inactive** need not be cleared when the utimer has reached zero in the current step (just after being decremented), but adding a special condition for that would increase code size as well as time complexity of the service for the very modest advantage of an occasional elimination of one unnecessary (and ignored) tick.

Having serviced the utimers, the function executes whatever snippets of code are provided in **irq_timer.h** (page 17). Macro **TCI_MARK_AUXILIARY_TIMER_ACTIVE** used by those snippets is defined as:

```
#define TCI_MARK_AUXILIARY_TIMER_ACTIVE \
      aux_timer_inactive = 0
```

in **MSP430/mach.h**. If the flag is found to be nonzero at the end of the service chain, the function masks out the interrupts from the respective CCR thus disabling the clock. Any system action that creates a new demand for the clock[5] will execute this operation:

```
#define TCI_RUN_AUXILIARY_TIMER tci_run_auxiliary_timer ()
```

to re-enable the event. The reason why it is defined through a macro is that the operation has to be redefined for the different implementations of the clock events. In particular, the "old" implementation (selected by setting **TRIPLE_CLOCK** to 0) renders that operation void (the strober clock is never disabled under that option).

Here is how the strober clock is restarted after a possible period of dormancy (**MSP430/main.c**):

```
void tci_run_auxiliary_timer () {
      cli_aux;    // in case the clock is not dormant
      TCI_CCA = gettav () + TCI_HIGH_DIV;
      sti_aux;
}
```

The respective CCR is simply assigned the current value of the counter + 4 ticks setting the clock to go off one millisecond from now. The clock's interrupt is also enabled. The current value of the counter is read by the function **gettav** with does it in a moderately tricky way. As the counter is being constantly updated by a different oscillator than the one clocking CPU instructions, reading it requires some care (because it may change literally *while* being read). This is how the problem is mitigated (**MSP430/main.c**):

```
static word gettav () {
```

[5] For illustration, see file **PicOS/irq_timer_leds.h** containing the code for blinking LEDs.

```
        word del;
        while (1) {
              del = TCI_VAL;
              if (TCI_VAL == del && TCI_VAL == del &&
                 TCI_VAL == del)
                    return del;
        }
}
```

The function implements a flavor of *majority vote* by reading the counter four times and insisting that all four readings produce the same value. The compiler will not optimize out that (apparently nonsensical) code as the memory location of the timer counter is declared as **volatile**.

The seconds clock, i.e., the leftover part of the interrupt service function (after the **if** statement) is much simpler. For one thing, the seconds clock never stops. At the very beginning of service, the CCR is incremented by **TCI_SEC_DIV** (4096) to schedule the next interrupt exactly one second after the current one. The seconds counter (**__pi_nseconds**) is then advanced. Following that, any snippets brought in by **second.h** (see page 18) are run. If the system has been compiled with **STACK_GUARD == 1**, the macro **check_stack_overflow** expands into a test whether the special bit pattern stored in the last location allocated to the stack has not been overwritten (see **MSP430/mach.h**). That assertion is verified every second.

The delay clock uses a separate interrupt service function listed below.

```
static word setdel;
...
interrupt (TCI_VECTOR) timer_int () {
      cli_tim;
      __pi_new += setdel;
      setdel = 0;
      RISE_N_SHINE;
      RTNI;
}
```

Variable **setdel** stores the last delay interval (in milliseconds) that the clock was set for. Having gone off, the clock disables itself (it has to be set explicitly for each new delay), adds the delay that has just elapsed to **__pi_new** (see page 13), and clears **setdel** as a way of indicating that its job has been accomplished. Then it kicks the scheduler (such that **update_n_wake** will be run) and exits.

Recall that the first action performed by **update_n_wake** (page 14) is to execute **TCI_UPDATE_DELAY_TICKS** in order to make sure that NEW (**__pi_new**) matches the current indication of the hardware clock. Normally, when the function is run immediately after the event signaled by the clock (the scheduler has just been kicked by **RISE_N_SHINE**), **__pi_new** is guaranteed to show the right value. Note, however, that **update_n_wake** can be executed in other circumstances, e.g., there has been a non-timer event, or some process has incurred a non-trivial lag preceding the present iteration of the scheduler loop. Note that the function is also invoked when a process issues a delay request (page 16). In principle, it may run at any moment after the delay clock was set and before it goes off. In such a case, it has to determine how much of the currently running delay interval has actually elapsed.

The requisite operation is defined as this macro (**MSP430/mach.h**):

```
#define TCI_UPDATE_DELAY_TICKS \
      tci_update_delay_ticks ()
```

We have the same problem as in the case of **TCI_RUN_AUXILIARY_TIMER**: we need a macro because the operation is void in the "old" implementation (**TRIPLE_CLOCK == 0**), where the delay clock ticks constantly at millisecond intervals, which means that NEW is always up to date (with the accuracy of 1 millisecond).

The actual code is here (**MSP430/main.c**):

```
void tci_update_delay_ticks () {
      cli_tim;
      if (setdel) {
            __pi_new += setdel -
                  TCI_TICKSTODEL (TCI_CCR - gettav ());
            setdel = 0;
      }
}
```

The function stops the timer and then looks at **setdel**. If **setdel** is zero, it means that the clock has not been running,[6] i.e., NEW (**__pi_new**) is up to date the way it looks.[7] Otherwise, the function calculates for how long the clock has been running since it was set by subtracting from **setdel** the difference between the target value in the CCR and the current indication of the counter. That difference must be converted to milliseconds, i.e., divided by 4 (or shifted two bits to the right) which is accomplished by the macro **TCI_TICKSTODEL**. Then **setdel** is zeroed to mark the fact that the timer has produced the delay (albeit prematurely) and thus completed its job. It will be set again from **update_n_wake** with the following operation:

```
#define TCI_RUN_DELAY_TIMER tci_run_delay_timer ()
...
void tci_run_delay_timer () {
      word d;

      cli_tim;
      // Time to elapse in msecs
      d = __pi_mintk - __pi_old;
      // Don't exceed the maximum allowed
      setdel = (d > TCI_MAXDEL) ? TCI_MAXDEL : d;
      TCI_CCR = gettav () + TCI_DELTOTICKS (setdel);
      sti_tim;
}
```

The function calculates the required delay as the difference between MINTK (**__pi_mintk**) and OLD (**__pi_old**) – see page 13. If this value is bigger than the maximum possible setting of the timer (note that the timer's range is 1/4 of the **delay** range in milliseconds), the maximum legitimate setting is applied instead. Then, **setdel** is set to the requested delay in milliseconds, and the CCR register is set to the current value of the counter + the delay (which has to be converted to counter ticks, i.e.,

[6] Note that masking the interrupt also prevents a race condition with the interrupt service routine.

[7] Note that there is no need to worry about the validity of NEW if no process is waiting on **delay**. If some process issues a **delay** request while no other process is waiting, the current setting of NEW (which should be equal OLD at this moment) can be safely used as the reference point - whatever it is.

multiplied by 4). Finally, the clock is enabled to trigger an interrupt when the CCR target has been reached.

The legacy option

On those MSP430 boards where the primary crystal of the CPU is high-speed (and also on eCOG1), the three system clocks are implemented using a single interrupt occurring regularly at millisecond intervals. This option can also be set forcibly for other boards by setting **`TRIPLE_CLOCK == 0`** at compilation. In that case, the hardware timer (on MSP430 it is still Timer B or the first Timer A) is configured to run in the so-called *up* mode, where it counts to the value in CCR0, triggers an interrupt, and automatically resets to run from zero. This way, once CCR0 is set, it doesn't have to be touched again.

The timer interrupt function advances NEW and also implements the strober clock. As the strober clock never stops, the macros **`TCI_MARK_AUXILIARY_TIMER_ACTIVE`** and **`TCI_RUN_AUXILIARY_TIMER`** are both declared as no-ops.

The seconds clock is implemented in **`update_n_wake`** with this extra code:

```
#if TRIPLE_CLOCK == 0
      millisec += (znew - __pi_old);
      while (millisec >= JIFFIES) {
            millisec -= JIFFIES;
            __pi_nseconds++;
            check_stack_overflow;
#include "second.h"
      }
#endif
```

which should be inserted into the sanitized version shown on page 14 right after the label **`MOK`**. Variable **`millisec`** accumulates ticks towards a second (**`JIFFIES`** is 1024) and **`__pi_nseconds`** is incremented on every full second. Al the other actions performed every second, like running the snippets defined in **`seconds.h`** as well as checking the stack against overflow, are carried out from there as well.