Olsonet Communications

PicOS

Version 3.0
May, 2010

Table of contents

1 Preamble

PicOS is a small-footprint operating system for organizing multiple activities (threads) of embedded *reactive* applications (praxes) executed on a small CPU with limited resources. It provides a flavor of multitasking (implementable within very small RAM) as well as simple and orthogonal tools for inter-process communication.

As of the time of writing, PicOS has been implemented on two CPU architectures: eCOG1 of Cyan Technologies[1] and the MSP430 CPU series of Texas Instruments.[2]

PicOS Virtual NETwork Interface (VNETI) is a sister document, referred to as [VNETI]. Other documents, describing various hardware-specific features of the system, will accompany this writeup.

An important component of the entire (holistic) platform based on PicOS is VUEE (or VUE^2) - the Virtual Underlay Execution Engine, which is also described in a separate document. VUE^2 makes it possible to recompile a PicOS praxis for its virtual execution in an emulated multi-node environment. For that, the praxis must be programmed with certain restrictions that render it VUE^2-compliant.

2 Introduction

The primary problem with implementing classical multitasking on micro-controllers with limited RAM is minimizing the amount of fixed memory resources that must be allocated to a process. The most critical example of such a resource is the stack space. Namely, under normal circumstances, every task must receive a separate and continuous chunk of memory usable as its private stack. Even if the stack size per task is drastically limited, it still remains a significant component of the total amount of memory resources needed to describe and sustain a single task. With the 2K words of on-chip memory available on the eCOG1, or 2KB of the typical RAM size for MSP430, this problem makes it very difficult to implement classical multitasking involving any non-trivial number of processes. This is because the stack space is practically wasted from the viewpoint of the application: it is merely a *working storage* needed to build the high-level structure of the program, and it steals the scarce resource from where it is truly needed, i.e., for building the *proper* (global) data structures.

PicOS solves this problem by adopting a non-classical flavor of multitasking. The different *tasks* share the same global stack and act as co-routines with multiple entry points and implicit control transfer. A task looks like a finite state machine (FSM) that changes its states in response to events. The CPU is multiplexed among the multiple tasks, but only at state boundaries. This simplifies (to the point of practically eliminating them) all synchronization problems within the application, while still providing a reasonable degree of concurrency and responsiveness. Besides, by enforcing the FSM appearance of a task, PicOS stimulates its clarity and self-documentation, which is especially useful and natural for reactive applications, i.e., ones that are not CPU bound, but perform finely-grained operations in response to possibly complicated configurations of events.

PicOS inherits its programming paradigm from SMURPH (also called SIDE in its most recent version), which was intended as a programming language for describing reactive systems (mostly telecommunication systems) for the purpose of constructing their high-fidelity simulation models. At some point, it was noticed that an elaborate simulation model expressed in SMURPH looked like an actual description of the *real thing*, which

[1] See www.cyantechnology.com.
[2] See http://www.ti.com/.

could be directly *compiled* into a true physical implementation. This hinted at the possibility of using the simulation language, along with the underlying run-time semantics of multiple threads, to program real systems, rather than their models. PicOS can be viewed as a successful verification of this idea in the micro-controller domain. Its present version supports several complex applications (praxes), including ad-hoc wireless communication systems implementing sophisticated and highly efficient routing schemes.

The fact that PicOS is closely related to SIDE makes it possible to emulate PicOS programs in a realistic virtual environment created in SIDE. This has brought us VUE[2] (Virtual Underlay Emulation Engine), which is a SIDE-based emulator for wireless praxes programmed in PicOS.

3 Processes

A process in PicOS is described by its *code* (which looks like a function with multiple explicitly declared entry points) and *data* (representing the object on which the process is supposed to operate). The entry points of a process function are called *states*.

Processes are identifiable by integer numbers, dubbed *process identifiers*, assigned at the moment of their creation. A process is created explicitly (by another process) and can be terminated either by itself or by another process. One process, called *root*, is started automatically by the system after reset. This process must be provided by the praxis[3] and is responsible for creating all other user processes. Some processes, e.g., ones needed by device drivers, may be started internally by the system before the root process.

```
process (sniffer, sess_t)
    char c;
    entry (RC_TRY)
        data->packet = tcv_rnp (RC_TRY, efd);
        data->length = tcv_left (packet);
    entry (RC_PASS)
        if (data->user != US_READY) {
            when (&data->user, RC_PASS);
            delay (1000, RC_LOCKED);
            release;
        }
        ...
    entry (RC_LOCKED)
        ...
    entry (RC_ENP)
        tcv_endp (data->packet);
        signal (&data->packet);
        proceed (RC_TRY);
endprocess(0)
```

Figure 1. Sample process code in PicOS.

[3]This is PicOS's term for “application.”

A process description starts with the **process** statement (see Figure 1), which takes two arguments. The first argument identifies the process, i.e., names the function that will be called whenever the process is run. The second argument describes the type of the process's local data object, i.e., the particular data structure that will be specific to the process instance. This makes sense when there are multiple instances of the same process: in such a case, each instance will operate on its own local data, in addition to being able to access all global data shared by all processes. For the process shown in Figure 1, the data type is **sess_t**. This means that whoever creates the process will have to pass it a pointer to an object of type **sess_t** (a variable of type **sess_t***). This pointer will be presented to the process in the local (implicitly declared) variable **data** whose type is **sess_t***.

The closing statement of a process definition is **endprocess**. Its argument specifies the limit on the number of instances of the process that can be run simultaneously, with 0 standing for "no limit." An attempt to create a process instance exceeding the limit will softly fail.

Some processes, mostly those that never run in more than one copy, may need no local data object: they will operate solely on global data. In such a case, the second argument of **process** can be **void**. PicOS introduces a terminology and operations for distinguishing between these two types of processes, i.e., the ones that need local data objects and the ones that do not. Processes of the first type are called *strands*, while the processes belonging to the second class are dubbed *threads*. For example, the process from Figure 1 is a strand and can be encapsulated as shown in Figure 2, where the dots stand for the original content. In this case, the **strand** statement is exactly equivalent to **process** (it is simply a straightforward alias), and **endstrand** is synonymous with **endprocess(0)**, which imposes no explicit limit on the number of coexisting copies of the process. Similar operations, **thread** and **endthread**, can be used to define a thread-type process, i.e., one that requires no local data. The **thread** statement takes a single argument (the process function name), and **endthread** is equivalent to **endprocess(1)**. The latter reflects the fact that a process taking no local data usually exists in a single copy only.

```
strand (sniffer, sess_t)
  ...
  ...
endstrand
```

Figure 2: A strand-type process.

Formally, the differentiation between threads and strands is not needed: the extra operations provide simple aliases for **process** and **endprocess**.[4] However, consistent adherence to the two aliases, and avoidance of the (more basic) **process** and **endprocess** statements, helps to make a PicOS praxis VUE[2]-compliant

The multiple entry points (states) of a process are identified by integer numbers between 0 and 4095 inclusively. These numbers are typically represented by symbolic constants assigning mnemonic names to the states (see Figure 1). Whenever a process is run (gets hold of the CPU), its code function is called at the *current* state, i.e., at one specific entry point. Initially, when the process is run for the first time after its creation, its code

[4] Entering a thread is marginally cheaper than entering a process, as the local variable **data** need not be set in the former case.

function is executed at state 0. Thus, state 0, which must always be defined, is the initial state. In particular, on of the state symbols in Figure 1 (probably **RC_TRY**) should be defined as 0.

When it is run, a process may issue *wakeup requests* identifying the conditions (e*vents*) to resume it in the future, and associating states with those events. For example, the process in Figure 1 issues two such requests in state **RC_TRY**: one with the **when** operation (for a signal-type event to be triggered by another process), the other with **delay** (in this case the event will be triggered by a timer after the specified number of milliseconds). At some point the process puts itself to sleep. This happens when it executes **release** (as in state **RC_TRY** in Figure 1) or when it exhausts the list of statements in its last state (attempts to fall through **endprocess** or **endstrand**). Note that **entry** statements do not provide boundaries, in particular, the sniffer process in Figure 1 may enter the sequence of statements in state **RC_PASS** by falling through from **RC_TRY**.

A process that has gone to sleep (sometimes we shall say that such a process has completed its current state) will be run again when any of the possibly multiple events awaited by the process occurs – at the state that was associated with that event by the corresponding wakeup request. Note that a sleeping process may be waiting for more than one event – as the sniffer in state **RC_PASS**. In such a case, the process will be resumed by the *first* occurrence of *any* of those events. In particular, the order in which the corresponding wakeup requests were issued is immaterial. All such requests executed by the process in a given state (before the process goes to sleep) aggregate into an unordered collection of awaited events.

Whenever a process is resumed and run, its collection of awaited events is cleared, i.e., the previously issued (but not triggered) wakeup requests are erased. If the process decides to wait for the same or a similar configuration of events, it must reissue the respective wakeup requests.

If a process goes to sleep without issuing a single wakeup request, it will be resumed immediately in its last state. Such an operation can be viewed as a "go to" to the beginning of the current state with a side effect described below.

From the moment a process is resumed until it goes to sleep, it *cannot* lose the CPU to another process (although it may lose the CPU to the kernel, i.e., an interrupt). The CPU is multiplexed among the multiple processes exclusively at state boundaries. The idea is that whenever a process goes to sleep, the scheduler is free to allocate the CPU to another (available) process that is either not waiting for any event, or whose at least one awaited event is already pending.

The fact that all processes share the same stack requires from the programmer to exercise care with automatic variables (like, for example, **c** in Figure 1). Such variables will not survive state transitions, and they cannot be used to store anything across states, i.e., whenever another process can conceivably run in the meantime. This also applies to some operations that at first sight look innocent, e.g., **proceed**, which has the appearance of a straightforward “go to”. In fact the operation passes through the scheduler to give it a chance to run another process that may have become ready. A variable whose contents must be preserved across state transitions must be either declared as static or become part of the process’s private data area (as described further in this document).

Owing to the fact that a process in PicOS requires no implicit memory resources, the amount of kernel memory needed to describe a process is determined solely by the size

of the Process Control Block (PCB). The exact size of the PCB is configurable at compilation, depending on the maximum number of events that a single process may want to await simultaneously, and is described by the following formula:

$$PCB\ size = F + 2E\ words,$$

where F = 5 for eCOG1 and 4 for MSP430, and E is the maximum number of events. With the default value of E = 4 (which so far has proved sufficient for all our praxes), this formula yields 13 words (or 26 bytes) for eCOG1 and 12 words (24 bytes) for MSP430.[5]

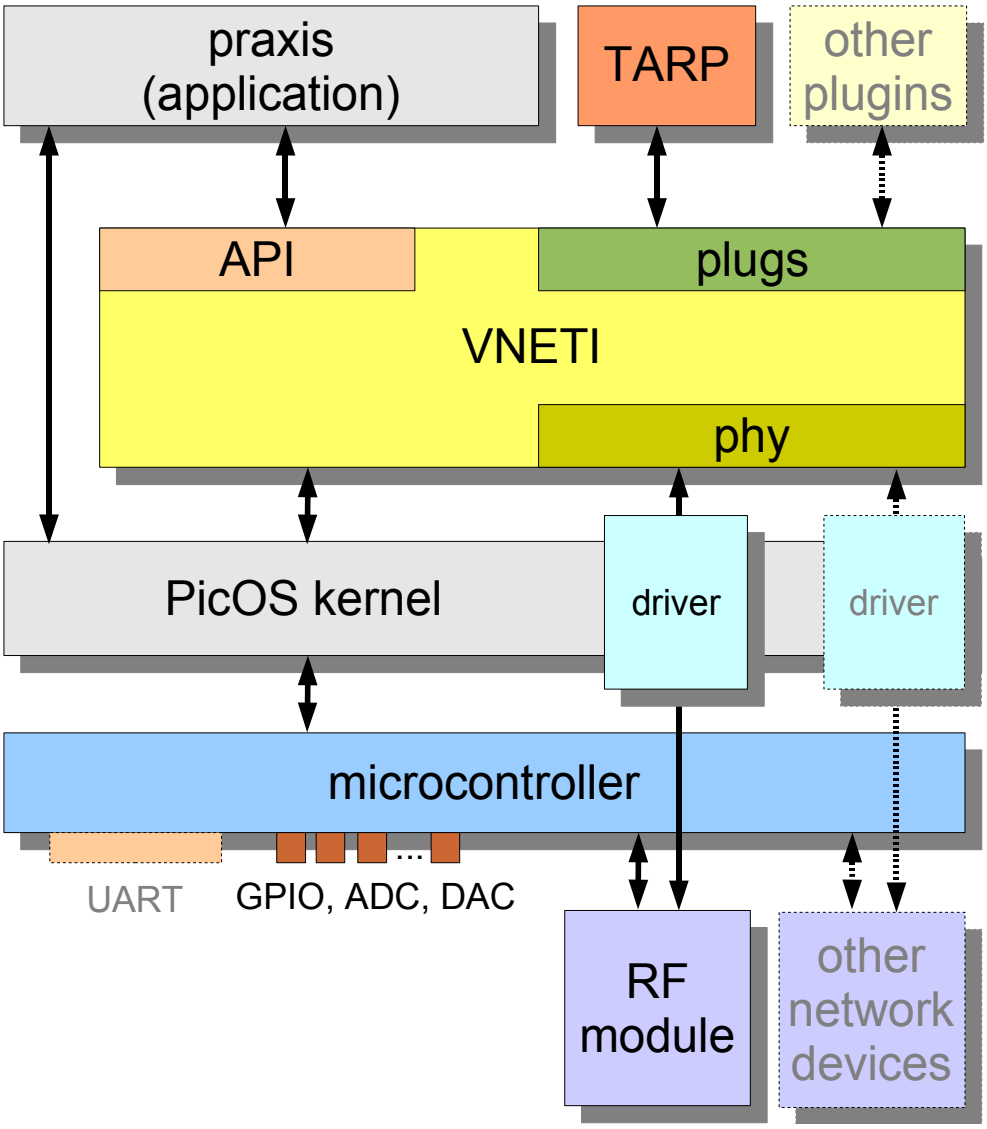

Figure 3. System structure.

[5] As PicOS was originally developed on eCOG1, where byte addressing is different from word addressing, it uses the (2-byte) word as the preferred basic unit of memory.

4 System organization

PicOS can talk to several I/O devices, e.g., the UART (possibly more than one) and RF modules. Usually, an RF module is not directly visible as an I/O device, but is interfaced to the praxis via VNETI, which comprises an orthogonal set of uniform session management tools. The functionality of VNETI (e.g., in terms of the routing/transport protocols) is extensible via *plugins*. This way, networking software in PicOS is inherently layer-less. Other physical devices (besides RF modules) can also be made visible via VNETI. This may make sense for those devices that act as telecommunication interfaces (e.g., a UART connected to another computer, as opposed to a terminal).

The organization of PicOS is shown in Figure 3. Note that the functionality of VNETI and its three types of interfaces (API, plugs, and phy) are described elsewhere.

A PicOS application (praxis) is typically intended for a specific *board*. The system includes an extensible set of board descriptions (for each of the two generic CPU types) specifying details like I/O pin configurations, RF module interface, UART parameters, etc., which depend on the layout of a particular board. New board descriptions can be created by modifying the existing board description files.

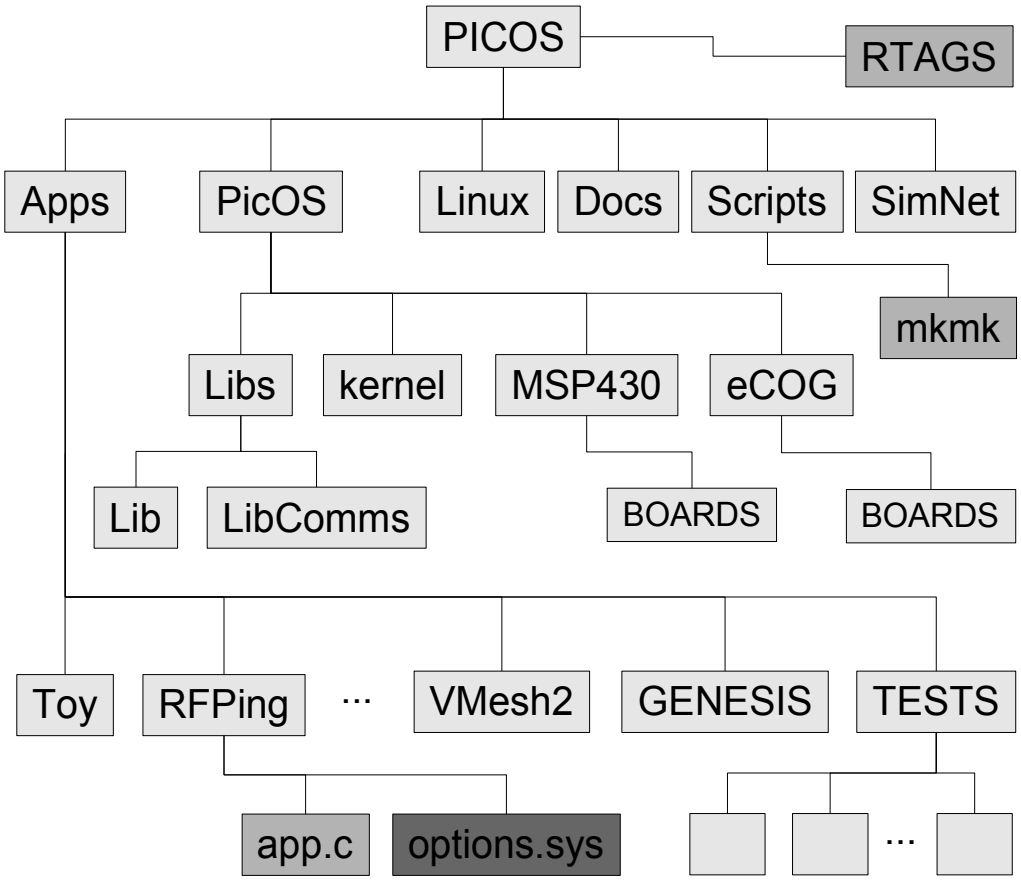

Figure 4. PicOS directory tree.

Figure 4 shows the (somewhat simplified) layout of PicOS directories. File *RTAGS* at level zero contains the history of modifications along with the tags identifying the respective versions of the system in the CVS repository. The main-level directories store the following components:

- *Apps*

  This is a repository for praxes, with each praxis occupying one sub-directory. Some directories in *Apps*, like *TESTS*, may themselves contain application directories. At least one file are required to describe an application: *app.c* containing the main part of the praxis program. The optional *options.sys* file may override some system parameters associated with the board for which the praxis is intended.

- *PicOS*

  This directory contains the source code of the system kernel, library modules, all device drivers, and the VNETI module. In addition to several C files containing various architecture-independent components of the system, the directory includes several subdirectories with:

  kernel — containing the architecture-independent kernel code, including the CPU scheduler.

  eCOG — containing the components pertinent to the eCOG1 CPU (including architecture-specific parts of device drivers).

  MSP430 — containing the components pertaining to the MSP430 CPU (including architecture-specific parts of device drivers)

  Libs — containing library modules partitioned into several subdirectories: *Lib* with various “utility” programs, e.g., implementing formatted I/O, as well as machine-independent drivers of RF devices, and *LibComms* with miscellaneous networking code (including the TARP plugin), and others.

  Each of the two CPU-specific directories, *MSP430* and *eCOG*, contains, among other things, a subdirectory called *BOARDS*, which in turn contain board specifications. Each specification is stored in a subdirectory of *BOARDS* whose name is interpreted as the board identifier. This identifier must be specified as an argument of *mkmk* (see below) when the target downloadable praxis image is compiled.

- *Linux*

  This directory contains simple programs developed under Linux for testing the Ethernet and serial interfaces of the eCOG board. Those programs can be used for developing an emulation server modeling the behavior of a radio channel.

- *SimNets*

  This directory contains the simulator project (in MS Studio) providing for virtual execution of PicOS praxes on the eCOG1 emulator.

- *Docs*

  Documentation directory.

- *Scripts*

  A few scripts in Tcl. One of those scripts, named *mkmk*, generates projects for PicOS praxes. The directory also includes file *UTILS.zip*, containing a packaged set of some handy tools, including the free variant of FET loader from Elprotronic and a terminal emulator.

The board description in a subdirectory of *BOARDS* consists of a number of files whose exact population depends on the configuration of system components. For example, if a radio module is interfaced to the system, the subdirectory will contain a file named *board_rf.h* specifying the pin interface to the module. One file that is always present there is *board_options.sys*, which defines a set of symbolic constants selecting various system components and options. The full list of those options together with their default values is present in file *options.h* in directory *PicOS*. The praxis may define additional options (not selected in *board_options.sys*) or override some of of the options mentioned in *board_options.sys*. This is rare and usually concerns only some "soft" options (e.g., debugging). It would make no sense for the praxis prepared for a specific board to redefine the board's hardware (in such a case, a new board definition would be provided instead). From now on, when we use the term "system option" or "system parameter," we shall mean a member of the set of system configuration options collectively determined by *board_options.sys* in the respective board configuration directory and *options.sys* in the praxis directory.

5 Software co-requisites

The following software (available for free) is required to compile, load, and debug PicOS praxes under Windows. The mspgcc MSP430 development toolkit can also be used under UNIX (Linux); however, the eCOG compilation platform, which requires CyanIDE from Cyan technologies, only works under Windows. In the sequel, we restrict ourselves to Windows.

Cygwin (see http://www.cygwin.com/)

Strictly speaking, the Cygwin environment is not absolutely required for eCOG development, although it is very useful. For example, the *mkmk* script generating project files for CyanIDE requires Tcl, which is immediately available under Cygwin.

CyanIDE Version 1.2 or later (see http://www.cyantechnology.com)

This is the official development platform for eCOG applications. It is not required for MSP430.

mpsgcc (see http://mspgcc.sourceforge.net)

This is the GNU development environment for MSP430 (not needed for eCOG). It requires Cygwin as a prerequisite.

File *UTILS.zip* in *PICOS/Scripts* contains the Lite FET-Pro430 JTag loader from Elprotronic (see http://www.elprotronic.com) and a terminal emulator. The terminal emulator requires no installation, but the FET loader must be installed. Terminal is a

convenient and powerful terminal emulator for interfacing UARTs to the PC. This software is also available from the network.

Please read the documentation that comes with CyanIDE and/or mspgcc. The rest of this document assumes that you are familiar with the technical details involved in interfacing a microcontroller board to the PC and debugging a program. The mspgcc environment is based on standard GNU tools, including gcc (as the compiler) and gdb (as the loader/debugger).

6 Quick start

6.1 eCOG

Connect the eCOG evaluation board to the PC as prescribed by the eCOG Quick Start manual. Make sure that UART A is connected to the PC, and a terminal emulator is attached to the corresponding COM port. Set the parameters of the port to 9600 bps, 8 bits, no flow control.

Move *mkmk* from *PICOS/Scripts* to a place where you keep private executables under Cygwin. Alternatively, include *PICOS/Scripts* in your $PATH.

Open a Cygwin window and move to *PICOS/Apps/Toy*. This directory contains a toy praxis, which illustrates the operation of three devices: the serial port, the LCD display, and the LEDs. Execute *mkmk CYAN* in that directory. This will create a CyanIDE project file named *Toy.cyp*. Note that *CYAN* specified as the argument to *mkmk* identifies the CYAN evaluation board. That board is described in subdirectory *CYAN* in *PicOS/eCOG/BOARDS*.

Click on the *Toy.cyp* file created by *mkmk*. This will open a CyanIDE project containing all the source files needed by the praxis. Click *Build* in the *Build* menu. The project will be compiled creating a file named *Image*.

Load the *Image* file into the board, e.g., by clicking *Download* in the *Debug* menu. Then click *Attach* (in the same menu) to start the praxis. You should see in the terminal window a text similar to this:

```
PicOS v2.4, Copyright (C) Olsonet Communications, 2002-2010
Leftover RAM: 1357 words
91C111 protocol type: 6006
91C111 MAC address: 0050c2187031
91C111 chip revision: 3391

Welcome to PicOS!
Commands:    'm ...anytext...' (LCD)
'b ...hexdigits...' (LEDs)
's bufsize nkwords' (SDRAM test)
'h' (HALT)
'r' (RESET)
```

The lines starting with 91C111 refer to the Ethernet chip. (Your MAC address is likely to be different.)

The program accepts a few commands, which are mostly used for testing and their repertoire tends to change with subsequent releases of PicOS. The letter *m* followed by an arbitrary string of characters (up to the nearest line feed) is treated as a message to be displayed on the LCD. If that string is 32 characters or less, it is displayed statically

(the capacity of the LCD display is 32 characters). Otherwise, it is periodically scrolled through the upper line of the display. Every new *m* command erases the previous message and replaces it with a new one. Another simple command is the letter *b* followed by a sequence of hexadecimal digits. Those digits determine the series of patterns to be displayed in the four LEDs. The patterns are switched periodically, and the sequence wraps around at the end. Similar to *m*, every new *b* command erases the previous pattern and replaces it with a new one.

6.2 MSP430

Connect the JTag interface of your MSP430 board[6] to the PC via a parallel cable. Also, make sure that the UART is connected to the PC, and a terminal emulator is attached to the corresponding COM port. Set the parameters of the port to 9600 bps, 8 bits, no flow control.

Move *mkmk* from *PICOS/Scripts* to a place where you keep private executables under Cygwin. Alternatively, include *PICOS/Scripts* in your $PATH.

Open a Cygwin window and move to *PICOS/Apps/Toy*. Alternatively, you can go to *PICOS/Apps/TESTS/ToySimple*, which is exactly the variant of Toy discussed as the programming example below. Execute *mkmk GENERIC_MSP430* in that directory. This will create a *Makefile* to turn the program into a loadable file. Note that *GENERIC_MSP430* names a generic MSP430 board (described in directory *PicOS/MSP430/BOARDS/GENERIC_MSP430*), which should be fine for all MSP430x1xx boards using the standard UART.

Execute *make*. This should compile the program creating two relevant files (different formats of the same loadable image): *Image* and *Image.a43*. The first file is in ELF format and can be loaded using the tools that came with mspgcc (msp430-gdb or msp430-jtag). *Image.a43* is in the so-called Intel HEX format and can be loaded with MspFet. Click on MspFet to start the program, then click *Erase* to remove any previous program from the board. Click *Open* and browse to *Image.a43*. Then click *Program* to load the file into the microcontroller.

If everything worked as needed, you will see something like this on the terminal:

```
PicOS v2.4, Copyright (C) Olsonet Communications, 2002-2010
Leftover RAM: 1227 bytes

Welcome to PicOS!
Commands:     'f %d %u %x %c %s %ld %lu %lx' (format test)
              'v' (show free memory)
              'r' (reset)
              'h' (halt)
```

There isn't a lot you can do with a simple board devoid of any devices except for UART. The *f* command tests formatted output, *v* reports on free memory, including the unused portion of stack, *r* resets the microcontroller (restarting the praxis), and *h* halts it.

6.3 A programming example

We shall discuss here the implementation of the Toy praxis for MSP430. Its complete source can be found in *PICOS/Apps/TESTS/ToySimple*. Let us start from the root process (thread), which plays the role of the main program:

[6] This quick start should work for any MSP430x1xx board equipped with JTag interface and a standard UART.

```
#define RS_INIT         0
#define RS_SHOW         1
#define RS_READ         2
#define RS_CMND         3
#define RS_FORMAT       4
#define RS_SHOWMEM      5

thread (root)
  static char *ibuf;

  entry (RS_INIT)
        ibuf = (char*) umalloc (IBUF_SIZE);
  entry (RS_SHOW)
        call (output, "\r\n"
          "Welcome to PicOS!\r\n"
          "Commands:\r\n"
          "       'f %d %u %x %c %s %ld %lu %lx' (format test)\r\n"
          "       'v' (show free memory)\r\n"
          "       'h' (HALT)\r\n"
          "       'r' (RESET)\r\n",
                RS_READ);
  entry (RS_READ)
        call (input, ibuf, RS_CMND);
  entry (RS_CMND)
        switch (ibuf [0]) {
          case 'f' : proceed (RS_FORMAT);
          case 'v' : proceed (RS_SHOWMEM);
          case 'h' : halt ();
          case 'r' : reset ();
        }
        call (output, "Illegal command or parameter\r\n", RS_SHOW);
  entry (RS_FORMAT)
        {
            int vi, q;
            word vu, vx;
            char vc, vs [24];
            long vli;
            lword vlu, vlx;

            vc = 'x';
            strcpy (vs, "--empty--");
            vli = 0;
            vlu = vlx = 0;

            q = scan (ibuf + 1, "%d %u %x %c %s %ld %lu %lx",
                        &vi, &vu, &vx, &vc, vs, &vli, &vlu, &vlx);
            form (ibuf,
                  "%d ** %d, %u, %x, %c, %s, %ld, %lu, %lx **\r\n",
                        q, vi, vu, vx, vc, vs, vli, vlu, vlx);
            call (output, ibuf, RS_READ);
        }
  entry (RS_SHOWMEM)
        form (ibuf,
              "Stack %u wds, static data %u wds, heap %u wds\r\n",
                    stackfree (), staticsize (), memfree (0, 0));
        call (output, ibuf, RS_READ);
endthread
```

This is straightforward C with a few confusing macros that appear like exotic keywords (**thread**, **entry**, **endthread**). Those macros are defined in file *sysio.h* in *PICOS/PicOS*, which must be included by any praxis program. The sequence of "defines" in front of the root process assigns symbolic names to the process's states.

As explained in section 3, the argument of the **thread** statement assigns a name to the process code function (e.g., to be referenced when the process is created). As root exists in a single instance (and needs no private data), it is a *thread* rather than *strand*.

When a process is run for the first time after its creation, its code function is entered at state number zero. The numbering of states does not have to follow any specific pattern (states can appear in any order and their numbers need not be consecutive), but the state number zero must be present somewhere within the process because otherwise it would not be able to start. Our root process starts in state **RS_INIT**.

The process code function resembles a regular C function and may include declarations of local variables. Let us repeat that any local (automatic) variables (allocated on the stack) do not survive state transitions; thus they cannot be used to store anything while the process is sleeping. Such variables can be used as scratch locations within a single state range, and perhaps it makes better sense to declare them in state blocks (as the set of variables in state **RS_FORMAT**) rather than immediately following the **thread** statement. Note that **ibuf** is declared as **static** and is thus *de facto* global.

For storing non-volatile data, a process has at least three options. Depending on the circumstances, it may use its "official" data object for this purpose, it may declare a static variable, or it may dynamically allocate a block of memory to accommodate its vital structures. As the root process never exists in more than one copy at a time, the second solution is a natural choice in this case.

In its initial state, the process allocates a buffer to be used for reading input lines and also for writing output to the UART. Similar to standard **malloc**, **umalloc** returns a chunk of memory with the specified size in bytes. Depending on the settings of system options, it is possible to have multiple and independent pools of heap memory, such that, for example, more critical components of the praxis need not compete for tight memory with less critical ones.

The next thing done by root, when in falls through to state **RS_SHOW**, is to write to the UART the menu of commands. This is accomplished by invoking **call**, which creates a special process to take care of this operation. The three arguments of **call** specify: the name of the process function, the value to be passed to the process as its data object pointer, and the state to be assumed by the caller when the called process terminates. In our case, we want the called process to act as a subroutine, with the caller waiting for its completion. The root process will not proceed to state **RS_READ** until the string passed to the **output** process spawned with **call** has been written to the UART.

In state **RS_READ**, root spawns another process in exactly the same way, this time to read a line from the UART. The **input** process referenced by the **call** statement will terminate when the input line has been stored in the buffer passed to it in the second argument of **call**. Then the root process will proceed to its next state **RS_RCMD**. In that state, the process looks at the first character of the input line, which is interpreted as a command. The cases **h** and **r** are immediately handled by built-in system functions. The remaining two commands are processed in separate states of root.

If the first character of the input line is not a valid command character, the process writes an error message to the UART. After the message has been written, root wakes up in state **RS_SHOW**, where it re-displays the command menu.

The **f** command is handled in state **RS_FORMAT**. The process parses the remaining portion of the input line looking for data items to be interpreted according to the format

string passed to **scan** (a library function). The number of items along with their values are then formatted (**form** is another library function) and written to the UART. After the output line has been written, the process wakes up in state **RS_READ** to process the next input command.

Similarly, in state **RS_SHOWMEM**, where root handles the **v** command, it formats an output line consisting of three numbers returned by some system functions and sends it to the UART. After the line has been written, the process returns to its main loop, i.e., state **RS_READ**.

We can now look at the code of the remaining two processes. Here is the one responsible for output:

```
#define OU_INIT         0
#define OU_WRITE        1
#define OU_RETRY        2

strand (output, char)
    static int nc;
    int k;

    entry (OU_INIT)
        nc = strlen (data);
    entry (OU_WRITE)
        if (nc == 0)
                finish;
    entry (OU_RETRY)
        k = io (OU_RETRY, UART, WRITE, data, nc);
        nc -= k;
        data += k;
        savedata (data);
        proceed (OU_WRITE);
endstrand
```

The output process is a *strand*. The type of its data object is **char**, which means that the data object pointer is of type **char***, which is a way of saying that the process receives a string pointer as the “thing” to work on. This string is available as **data**, which is a standard variable available to every strand.

Having started in state OU_INIT, the process determines the length of the output string, which is simply the number of characters to be written to the UART. State **OU_WRITE** marks the process’s “main loop,” in which it will try to send those characters until all of them have been accounted for. When this happens, i.e., **nc** runs down to zero, the process will terminate (by executing **finish**). In state **OU_RETRY**, the process invokes **io** to write some characters out. The function is PicOS’s standard way of accessing I/O devices. It tries to write **nc** characters pointed to by **data** to the device identified by its second argument (**UART**). If the operation cannot immediately succeed, the process is suspended until it makes sense to try again. Then, the process will be resumed in state **OU_RETRY** specified as the first argument of **io**.

Having returned (instead of temporarily blocking the process), **io** tells the number of characters that have been written to the device (it does not have to process them all in a single go). The process updates the data pointer and character count to reflect the progress and keeps looping for any remaining characters.

The reason for calling **savedata** is that the data pointer is intentionally constant. The process is free to change it within a given state, but when it loses the CPU and is

subsequently resumed, the data pointer will be reloaded (from the PCB) with its previous (original) value. This is because in principle it identifies the process's immutable data object: an area in memory whose location is not supposed to change. If the process knows what it is doing, it can modify its data pointer permanently by executing **savedata**, which function saves its argument as the new data pointer.

Here is the second auxiliary process, the one responsible for reading lines from the UART:

```
#define IN_INIT         0
#define IN_READ         1

strand (input, char)
    static int nc;
    int k;

    entry (IN_INIT)
        nc = IBUF_SIZE;
    entry (IN_READ)
        k = io (IN_READ, UART, READ, data, nc);
        if (*(data+k-1) == '\n') {
                // End of line
                *(data+k-2) = '\0';
                finish;
        }
        if (k > nc - 2)
                k = nc - 2;
        nc -= k;
        data += k;
        savedata (data);
        proceed (IN_READ);
endstrand
```

The process attempts to read a string of characters from the UART into its data object, which, similar to the **output** process, is a character array. The size of this array (and the maximum number of characters that can fit) is **IBUF_SIZE**. Each successful invocation of **io** reads up to **nc** characters into the array, with **nc** decreasing with every read. The last two locations in the buffer are reserved for the line terminator, which is a carriage return character (**\r**) followed by new line (**\n**). Having located the terminator (recognized by its second character **\n**), the process substitutes it with the null byte and terminates itself. Note that the updated data pointer is saved after every partial read.

To turn the above pieces into a complete praxis, you can put them into a single *app.c* file and insert the following two statements at the beginning:

```
#include "sysio.h"
#include "form.h"
```

The first file is the mandatory header for all PicOS praxes, while the second one contains the declarations of the formatting functions **scan** and **form**. This *app.c* file can be compiled for the generic board in the way described above. You may want to go to *PicOS/MSP430/BOARDS/GENERIC_MSP430* and have a look at the files in that directory. One of those files, *board_options.sys*, lists the system options that are switched on for the particular board. One non-standard feature needed by our simple praxis is the function **memfree** (called from root) which is only available if the option **MALLOC_STATS** is switched on. As this option is not set in *board_options.sys*, it must be defined in the *options.sys* file associated with the praxis (and stored in the same

directory as *app.c*). Thus, the last item to complete the application is file *options.sys* with the following contents:

```
#define MALLOC_STATS            1 /* Malloc statistics */
```

The full list of system configuration options with default settings and comments can be found in *options.h* in directory *PICOS/PicOS*.

7 Basic types

All PicOS-specific and user-visible types, functions, macros and constants provided and understood by the system are defined in file *sysio.h,* which references other relevant header files and must be included by any praxis integrated with PicOS. In addition to the standard types defined by C, PicOS defines the following types:

word	equivalent to	**unsigned int**
lword	equivalent to	**unsigned long int**
address	equivalent to	**word***
byte	equivalent to	**unsigned char**
bool	equivalent to	**unsigned char**

The following symbolic constants define the ranges of the relevant arithmetic objects:

```
#define MAX_INT             ((int)0x7fff)
#define MAX_UINT            ((word)0xffff)
#define MIN_INT             ((int)0x8000)
#define MIN_UINT            ((word)0)
#define MAX_LONG            ((long)0x7fffffffL)
#define MIN_LONG            ((long)0x80000000L)
#define MAX_ULONG           ((lword)0xffffffffL)
#define MIN_ULONG           ((lword)0)
```

Please consult these files: *sysio.h, options.h, mach.h, arch.h* (the last two located in the respective CPU-specific directories: *eCOG* and *MSP430*) for more constants and macros, as well as for the global configuration parameters of the system. Specific configuration parameters, settable on a per-board basis, are stored in *board_options.sys* in the respective subdirectories of *BOARDS*.

8 Essential operations provided by the kernel

PicOS is simple enough to be studied and understood by examining the code of its sample praxes. Below, we list all the essential operations that the system makes available to its processes, in no particular order (or, perhaps, in the loose order of their relevance).

```
int fork (code_t code, address data)
```

This operation creates a new process and returns its identifier (*pid*). The new process executes concurrently with its creator. The first argument refers to the process code function (the first parameter of **process**), the second gives the pointer to the process data object. Note that this is a **word** pointer. Care should be taken when passing character pointers as data objects on the eCOG version of PicOS. Owing to the special way in which such pointers are treated by the eCOG CPU, they can only be properly cast to the **word** pointer type if they happen to be aligned at a **word** boundary.

If the new instance of the created process would violate the restriction on the number of its simultaneously active copies (declared with the argument of **endprocess**), **fork** returns zero and its action is void. A legitimate *pid* can never be 0.

In accordance with the distinction between threads and strands, there exist two aliases for **fork**, namely **runstrand (c, d)** is exactly equivalent to **fork (c, d)**, and **runthread (c)** translates to **fork (c, NULL)**.

int kill (int pid)

This function kills the indicated process. As a special case, **kill(0)** kills the current (invoking) process and is exactly equivalent to **finish**.[7] Another special case, **kill(-1)** (equivalent to **hang**), turns the caller into a *zombie*. This means that the caller effectively terminates itself (in particular, triggering a termination event perceived by **join**), but it doesn't disappear (retaining its *pid*), such that it still can be perceived as active by **running** (see below). The only way for a zombie to disappear altogether is to be killed by some other process. This kind of handshake, i.e., process A terminating itself and becoming a zombie until another process B has noticed its termination and killed A, is useful in certain IPC (inter-process communication) scenarios. None of **kill(0)**, **finish**, **kill(-1)**, **hang** ever returns. A returning **kill** returns the *pid* of the killed process, or 0 if the specified process wasn't found.

int killall (code_t code)

This function kills all process instances running the indicated code function and returns the number of processes that have been killed. If no process is running the specified code function, **killall** returns zero and has no effect.

int running (code_t code)
int crunning (code_t code)

The argument is a code function pointer. If an instance of a process using the specified code function is currently active, **running** returns the *pid* of one (any) such instance. Otherwise, the function returns zero. A zombie is considered "active" by this function. The second variant (**crunning**) returns the number of active processes (including zombies) using the specified code function.

If **code** is **NULL**, then **running** will return the *pid* of the current process (the one that has invoked the function), and **crunning** will count the number of unused process slots, i.e., the maximum number of processes that can still be created.

For compatibility with VUEE, it is recommended to use these aliases:

getcpid ()	for	**running (NULL)**
ptleft ()	for	**crunning (NULL)**

int zombie (code_t code)

If there is a zombie process using the specified code function, **zombie** returns the *pid* of one (any) such process. Otherwise, zero is returned.

[7] Note that **finish** is not a function but a macro, which appears as a parameter-less operator.

code_t getcode (int pid)

Returns a pointer to the code function executed by the indicated process (and can be viewed as an inverse of **running**). If **pid** is zero, the pointer to the current process's code function is returned.

int status (int pid)

Returns the number of events awaited by the indicated process or −1 if the process is a zombie. For example, *sysio.h* defines this macro:

```
#define iszombie (pid) (status (pid) == -1)
```

void when (word event, word state)

Issues a wakeup request for the indicated event. When the event is triggered, the process will be resumed in the specified state. Although **state** is formally 16 bits long, its value is truncated to the least significant 12 bits. A state number must be between 0 and 4095 inclusively. For compatibility with older versions of PicOS, this operation is also available as **wait**, i.e., **when (e, s)** is equivalent to **wait (e, s)**. The usage of **wait** is discouraged as it conflicts with VUE[2].

int trigger (word event)

Triggers the indicated event (awaited by **when**). Returns the number of processes that have been waiting for it.

For the purpose of **when** and **trigger**, events are identified by 16-bit numbers viewed as bit patterns with no implied semantics. Certain (special purpose) events can be awaited with some other operations (e.g., **join**, **io**, **delay**). Such events are never confused with those awaited by **when** and triggered by **trigger**.

Triggered signals are never queued: if no process is waiting for a triggered event, the **trigger** operation is ignored. On the other hand, if there are multiple processes awaiting for the same triggered event, all of them will be awakened.

int ptrigger (int pid, word event)

Like **trigger** (see above), except that the event is only triggered for the indicated process. If **pid** is zero, the operation is exactly equivalent to **trigger**. If **pid** is nonzero, but it does not point to an alive process, the operation causes an error. If there is only one process that can possibly await the event and its **pid** is known, **ptrigger** is more efficient than **trigger**.

The function returns the number of processes waiting for the event, which, with exception of the case **pid** = 0, can only be 0 or 1.

release

This is a parameter-less operation which leaves the current process, i.e., explicitly forces the completion of the current state. When executed from the process's body, it can be replaced by simple **return**. Note, however, that **release** can be executed from a function called from the process body, in which case it will exit the process rather than return from the function.

int join (int pid, word state)

Issues a wakeup request for an event to be triggered when the indicated process terminates (or becomes a zombie). The function has no effect if the process already is a zombie, or if it doesn't exist, and returns zero in such a case. Otherwise, it returns the first argument (**pid**).

int joinall (code_t code, word state)

Issues a wakeup request to be triggered when any process running the specified function terminates or becomes a zombie. In contrast to **join**, the request is also issued (i.e., the event does not occur immediately) if no process with the specified code function is currently running, or if some such processes are already zombies.

void delay (word timeout, word state)

Issues a timer wakeup request, i.e., sets up an alarm clock to go off after the specified number of *milliseconds.*[8] When that happens, the process will be restarted in the indicated state. At most one alarm clock per process can be set up at any time. If a process issues multiple **delay** requests before releasing the CPU, only the last of them is effective. A **delay** request is interpreted in a way similar to a **when** request and can naturally coexist with other wakeup requests.

word dleft (int pid)

This function returns the number of PicOS milliseconds that the indicated process will still sleep for on a **delay** timer (see above), i.e., remaining until the timer goes off (if no other event wakes the process up in the meantime). If the process is not waiting on a **delay** timer, or **pid** does not identify an existing process, the function returns **MAX_UINT**. The case **pid** == 0 is viewed as an inquiry about the current process and is special. If the issuing process has been waiting on a **delay** timer and has been awakened (by the timer or, possibly, some other event), such a call will return the residual number of milliseconds remaining for the timer to go off. Note that this number will be zero, if the process has been in fact awakened by the timer. The function will return garbage, if the current process didn't issue a **delay** request in its previous state.

lword seconds (void)

This function returns the number of seconds since the system was started.

void proceed (word state)

This operation unconditionally transits to the indicated state. The transition is different from a "go to" in that it involves the CPU scheduler. For example, if the current process executing **proceed** has triggered events waking up other processes, those other processes are given a chance to preempt the current process.

void call (code_t code, address data, word state)

This operation is exactly equivalent to **join (fork (code, data), state)** immediately followed by **release** (and is implemented as a macro). It creates a

[8] In PicOS, one millisecond is defined as 1/1024 of a second.

new process and puts the parent process to sleep until the child process terminates itself. When this happens, the parent is resumed in the indicated state.

`void savedata (void *data)`

The data object pointer associated with the current process is set to the specified value. Normally, the data pointer (variable **`data`**) available to the process is not expected to change during the process's lifetime. Even though the process can formally change the **`data`** pointer, the change is undone after every state transition because whenever the process is run, the data pointer is re-loaded from its original value stored in the PCB. With **`savedata`**, the process is able to set its **`data`** pointer to any value in a permanent fashion.

`int io (int state, int device, int op, char *buf, int len)`

This is the single general-purpose i/o operation. In the present version of the system, the implemented devices are **`UART`**, **`UART_A`**, **`UART_B`**, **`LCD`**, **`ETHERNET`**, (the last two devices exclusively on the eCOG), and the operations are **`READ`**, **`WRITE`**, and **`CONTROL`**. Note that RF modules are interfaced via VNETI, and data exchange with them does not involve **`io`**.

Warning: *this function may be removed in the future. It appears that using* **`io`** *for UART is an overkill (other, more efficient, ways have been implemented), and other device types are better handled by VNETI or special (dedicated) functions.*

The five arguments stand, respectively, for the blocked state identifier, the device number, the operation, the buffer pointer, and the buffer length. Generally speaking, the function attempts to perform the indicated i/o operation, with the parameters described by **`buf`** and **`len`**, on the **`device`**. Depending on the circumstances, the operation may succeed immediately (without blocking) or not. For **`READ`**/**`WRITE`**, the operation is considered successful if at least some (initial) bytes in the buffer have been transferred to/from the device. In such a case, **`io`** returns the number of processed bytes. Otherwise, the device is considered busy and the function *does not return*. The process is blocked and put to sleep awaiting an (internal) event that will be triggered when the device becomes available. When that happens, the process will be restarted in the state indicated by the first argument of **`io`**. The same idea applies to **`CONTROL`** operations, if it is possible for them to block. If an **`io`** operation cannot possibly block (in fact, most **`CONTROL`** requests never block) the first argument of **`io`** is irrelevant.

The way **`io`** awaits a status change on the device fits the general idea of (possibly multiple) wakeup requests issued by the process. Thus, if the process simultaneously awaits another event, and that event occurs earlier than the readiness of the i/o device, the process will be restarted by that event. This implies one possible way of issuing a non-blocking i/o request without spawning another process. Consider the following code fragment:

```
...
entry (SOME_STATE)

   delay (0, WOULDBLOCK);
   ptr += io (SOME_STATE, UART_A, WRITE, ptr, len);
   ...
entry (WOULDBLOCK)
...
```


Despite the fact that **io** issues an internal wakeup request and puts the process to sleep, the process also awaits a zero-duration timer delay, which event will be triggered immediately. Thus, if **io** decides to block, the process will transit to state **WOULDBLOCK**.

A simpler way to accomplish a similar thing is to use **NONE** (**-1**) as the state argument of **io**. In such a case, the function will never block, returning 0 when the device is busy. For operations other than **READ**/**WRITE**, any nonzero value returned by **io** should be viewed as an indication of success. A blocking **READ**/**WRITE** operation may legitimately return 0 when it is *void*. What that means generally depends on the device, but one standard case of a legitimate void **READ**/**WRITE** request is one specifying zero buffer length, i.e., zero bytes to transfer.

As a significant number of **io** requests tend to use **NONE** for the **state** argument, the system defines this macro:

```
ion (int device, int op, char *buf, int len)
```

as a shortcut for such occasions.

void utimer_add (word *utm)

This operation declares a countdown timer. The first argument points to a word variable which will be added to the global pool of countdown timers. As soon as its contents are set to a nonzero value (with **utimer_set** - see below), the word will be automatically decremented towards 0 at *millisecond* intervals. If the second argument is **NO** (zero), the indicated location is removed from the timer pool. If the first argument is **NULL**, the entire timer pool is cleared. The maximum size of the timer pool is 4 entries (and attempt to create more than 4 utimers will trigger a system error).

Note: If spare SDRAM[9] is used by the praxis, countdown timers cannot be located in (**malloc'ed**) SDRAM. This is because the operations of moving data to/from spare SDRAM temporarily re-map memory, which happens asynchronously with the timer interrupts that affect the counter. Consequently, a timer allocated in SDRAM could corrupt spare SDRAM storage during move operations.

void utimer_delete (word *utm)

This operation removes a utimer previously registered with **utimer_add**. The argument should point to a word variable. If the specified timer is not registered, the operation will trigger a system error.

void utimer_set (word utm, word val)

This operation sets the utimer variable **utm** to value **val**. Note that the first argument is the variable name (not a pointer). The operation is implemented as a macro.

udelay (word del)

This is a spin delay loop that takes approximately **del** microseconds.

[9] This applies exclusively to eCOG1.

mdelay (word del)

This is another spin delay loop that takes approximately **del** milliseconds. Do not overuse!

```
int strlen (const char*)
void strcpy (char *dest, const char *src)
void strcat (char *dest, const char *src)
void strncpy (char *dest, const char *src, int count)
void strncat (char *dest, const char *src, int count)
```

These are the standard operations on strings. They behave almost like the corresponding functions from UNIX except that the copy/cat functions return no values. Here are two more differences:

1. **strncpy** stops on a **NULL** byte, if encountered in the source string before the **count** runs out, and it also inserts the **NULL** byte at the end of the copied string (so that count + 1 bytes of the output string are overwritten);

2. **strncat** inserts the source string after count characters of the destination (overwriting the tail).

```
void memcpy (void *dest, const void *src, int count)
void memset (void *dest, int val, int count)
```

These functions operate like their UNIX equivalents: **memcpy** copies **count** (non-overlapping) bytes from **src** to **dest**, while **memset** sets **count** bytes starting from **dest** to **val**.

```
void diag (const char*, ...)
```

This function is intended for debugging. It writes a line of text directly to UART A bypassing the UART driver and interrupts. When the function returns, the line has been written, i.e., the operation is completely synchronous. The specified string is interpreted as a simplified print-like format with the following sequences considered special:

- **%d** the next **word**-sized argument of **diag** is fetched and encoded as a signed integer number;
- **%u** the next **word**-sized argument is fetched and encoded as an unsigned integer number;
- **%x** the next **word**-sized argument is fetched and encoded as a 4-digit hexadecimal number;
- **%s** the next **char***-sized argument is fetched and interpreted as a pointer to a character string to be inserted at this position.

```
void leds (int which, int how)
```

This function is used to switch on/off the LEDs available on the board. Generally, the number of LEDs is board-specific (some boards may have no LEDs at all). If present and available, the LEDs are numbered from 0 up to the total number of LEDs – 1. The first argument of **leds** is the LED number and the second specifies the action, which can be:

LED_off	**(0)**	the LED is turned off
LED_on	**(1)**	the LED is turned on
LED_blink	**(2)**	the LED is put into a blinking mode

The last option is only available if auto-blinking LEDs are enabled (by setting **LEDS_BLINKING** to 1 among system options).

void fastblink (bool on)

If auto-blinking LEDs are enabled, this function changes the blink rate to fast (if the argument is **YES**) or slow (if the argument is **NO**). The default rate is slow at approximately 2 blinks per second, while the fast rate is about 8 blinks per second.

void reset (void)

The microcontroller is unconditionally reset and the system is restarted.

void halt (void)

The microcontroller is unconditionally stopped.

void syserror (int code, const char *msg)

The function aborts the system presenting the specified error code and displaying the specified message. If the **DIAG_MESSAGES** system option is set to 2, system abort messages are verbose, and the message will be written to the UART. Otherwise, if **DIAG_MESSAGES** is less than 2, the message is ignored.

void sysassert (bool cond, const char *msg)

This is a macro which tests the condition specified as the first argument, and if it fails, executes **syserror (EASSERT, msg)**. **EASSERT** is defined as 10. The full list of errors generated by the system can be found in *sysio.h*.

word rnd (void)

This function returns an unsigned word-sized integer random number between 0 and 65,537. It is only available if the **RANDOM_NUMBER_GENERATOR** system option is greater than zero. Two versions of the random number generator are available. When **RANDOM_NUMBER_GENERATOR** is 1, the simpler (crude) version is selected, whose quality may be substandard from the viewpoint of its theoretical properties, which, however, should be more than adequate for typical applications (e.g., randomized backoff). With **RANDOM_NUMBER_GENERATOR** set to 2, the more advanced version is selected, which produces statistically better results at the cost of increased processing time and the overhead of 2 extra bytes of RAM.

Note that even the quality of the simpler version can be drastically enhanced with the addition of “external entropy” to the seed. When the **ENTROPY_COLLECTION** option is set to 1, the system will attempt to collect true randomness from some events (like RF reception parameters and timing) and store it in the **lword** variable **entropy**. This variable is global and directly usable by the praxis. It also automatically affects the generation of random numbers, which then become truly random (as opposed to being merely pseudo-random).

9 The UART

Up to two UARTs can be configured into the system and made available as devices number 0 (symbolic constant **UART_A**, or simply **UART**) and 1 (**UART_B**). Each of the two UART devices can independently operate in two different modes. In the standard (unlocked) mode, which is assumed by default, the operation of the device is driven by interrupts, and the semantics of **io** addressed to the UART conform to rules outlined above. Thus, **READ** reads the prescribed number of bytes from the UART, possibly blocking if not a single byte is immediately available for reception, and returning the number of read bytes, which can be less than the specified length. Similarly, **WRITE** attempts to send the indicated number of bytes to the device, possibly blocking if not a single byte can be immediately written, and returning the actual number of bytes that have been sent to the UART. Besides **READ** and **WRITE**, **CONTROL** operations can be used to change the UART's baud rate[10] or to switch the device to the so-called *locked* mode.

The following (never-blocking) command changes the baud rate of the UART:

```
ion (uart, CONTROL, (char*)&rate, UART_CNTRL_RATE)
```

The "buffer" argument should point to a 16-bit word (i.e., the type of **rate** should be **word**) storing the desired rate divided by 100 (i.e., 384 corresponds to 38400 bps). The assortment of legitimate rates depends on the crystal configured with the CPU. For an MSP430 using the standard low-power watch crystal at 32768 Hz, the available rates are: 1200, 2400, 4800 and 9600. With a high-speed crystal, additional four rates become available: 14400, 19200, 28800 and 38400. An attempt to set the rate to a value not on this list will trigger a system error.
Typically, UART parameters are set by system configuration options and there is no need to change them from the praxis. Here is the list of system options affecting the operation of UART together with their default values:

```
#define  UART_DRIVER                    0
```

Possible values: 0 = no UART present in the system, 1 = UART_A only, 2 = two UARTs, i.e., UART_A + UART_B.

```
#define  UART_RATE                      9600
```

Bit rate of the UART. If two UARTs are present, the rate is the same for both of them. However, if **UART_RATE_SETTABLE** is 1, the rate can be set individually for each UART from the praxis.

```
#define  UART_RATE_SETTABLE             0
```

If set to 1 (strictly speaking nonzero), the UART rate can be set from the praxis via **io CONTROL** (see above). If zero, the UART rate is fixed by **UART_RATE**. In the former case, **UART_RATE** only specifies the initial rate.

```
#define  UART_BITS                      8
```

[10] At present, the UART rate on eCOG cannot be changed (it is "hardwired" at 9600 bps). Thus, this fragment applies to MSP430 only. In the latter case, the possibility to dynamically modify the UART rate from the praxis is a system configuration option: **#define UART_RATE_SETTABLE 1**.

This can be 8 or 7 (the number of bits per character). Other values may work, but they are not guaranteed to.

```
#define  UART_PARITY                    0
```

Parity control, which can be 0 (for even) or 1 (for odd). This is ignored when **UART_BITS** is 8, which implies "no parity."

```
#define  UART_INPUT_BUFFER_LENGTH       0
```

Sets the buffer length for UART input. With the setting of 0 or 1, there is no internal buffer, which means that input characters are directly delivered into the user-specified buffer (parameter of **io**). This may cause problems at high rates and/or if the praxis process reading characters from the UART is not very attentive: if it tends to lag with **io** requests, characters may be lost. With **UART_INPUT_BUFFER_LENGTH > 1**, the characters are first read into the internal buffer, which provides a damping storage for data bursts. Of course, the buffer occupies memory, so it shouldn't be used unless needed. It is generally not needed for keyboard input.

```
#define  UART_INPUT_FLOW_CONTROL        0
#define  UART_OUTPUT_FLOW_CONTROL       0
```

These options are only meaningful if the UART is interfaced with the flow control lines, which is seldom the case (generally, microcontroller boards implement a simple two-wire UART interface). If set to 1, the options enable CTS/RTS flow control for UART communication.

```
#define  UART_TCV                       0
```

If nonzero, this implies that the UART is handled by VNETI, i.e., it does not appear as a regular device but as an RF module equivalent. Only one UART (UART A) will be defined in this case. This requires **UART_DRIVER** to be 0. Other UART parameters retain their meaning.

The purpose of the locked mode of UART (applicable when **UART_DRIVER** > 0) is to make it possible to use the UARTs for emulating naive serial devices. When put into the locked mode, the device operates in a persistent, immediate and interrupt-less manner. This is accomplished with the following operation:

```
ion (uart, CONTROL, &c, UART_CNTRL_LCK)
```

where **c** is a single character. If this character is nonzero, the UART device is put into the locked mode; otherwise, the default (unlocked) mode is resumed. While in the locked mode, **READ**/**WRITE** requests addressed to the UART never block. For **READ**, the operation retrieves into the buffer those characters that are available immediately and returns their number, possibly zero. The process is never suspended even if no input is immediately available. The driver process never reads ahead any characters.[11] **WRITE** on a locked UART behaves in a fashion similar to **READ**. Only those characters that can be accepted immediately (possibly none) are sent to the device, and **io** returns their number (which can be zero). The process is not blocked when no characters can be written (the **state** argument of **io** is ignored).

[11] In fact, no driver process is needed in the locked mode. Thus, it is terminated when the device is put into the locked mode and restarted when the unlocked mode is resumed.

10 Memory allocation

PicOS is equipped with a simple and effective implementation of **malloc**, which makes it possible to organize the dynamically available RAM (known as the heap) into a number of disjoint allocation pools. With SDRAM[12] present in the system, the heap is located entirely in SDRAM. Otherwise, it uses whatever on-chip RAM remains available after accommodating all static variables. Note that constants (declared with the **const** keyword of C) are allocated in code and take no RAM space.

Multiple memory pools are enabled by default. In tight memory scenarios, it may make sense to disable them by setting

```
#define  MALLOC_SINGLEPOOL        1
```

among the system options. If multiple pools are disabled, all memory acquisition functions described below take memory from the same global pool.

The praxis declares the partitioning of the available heap memory into pools with the following statement:

```
heapmem {p0, p1, ..., pn-1};
```

which is declarative and should appear in the data section of the program. This statement is not needed (and if present is void) when multiple memory pools are disabled. The specified integer values ***p0, p1, ..., pn-1*** must be all strictly positive and they should add to 100. Their number ***n*** determines the number of memory pools, with each pool receiving the specified percentage of available memory. Pool 0 is intended as the primary storage used by the praxis. If VNETI is configured into the system, pool 1 covers the packet storage of VNETI (and is not used for anything else). More pools can be defined by the praxis, but the praxis should never use pool 1 if VNETI is present.

PicOS provides the following two basic functions to carry out memory allocation/deallocation for the praxis:

```
address malloc (int pool, word size)
void free (int pool, address ptr)
```

with the first argument identifying the pool (starting from zero) and the second argument specifying the minimum size of the requested chunk of memory in bytes (not in words). The allocated chunk always consists of an even number of bytes necessarily aligned at a word boundary. Its size may be bigger than requested. If **malloc** cannot fulfill the request, it returns **NULL**. A chunk allocated from a given pool must be returned to the same pool. If the second argument of **free** is **NULL**, the operation is void.

If multiple pools are disabled, the above functions are defined without the first argument, i.e.,

```
address malloc (word size)
void free (address ptr)
```

[12] On the eCOG. This feature is described in a separate document.

If the praxis consistently uses **umalloc** (see below) instead of **malloc**, if can make itself independent of the pooling issues.

The following operation:

```
word actsize (word *chunk)
```

returns the actual size of a chunk allocated by **malloc**.

Normally, the memory block allocated by malloc is word-aligned. Some praxes (using long integers, for example) may require such a block to be always aligned at a long-word boundary (a multiple of four bytes). This can be enforced with the following system configuration option:

```
#define  MALLOC_ALIGN4            1
```

The following macros (defined in *sysio.h*) are recommended for allocating/deallocating memory from the praxis (at least in those straightforward cases when a single memory pool per praxis is sufficient):

```
#define    umalloc(s)    malloc ([0, ]s)
#define    ufree(p)      free ([0, ]p)
```

The first argument of **malloc** is only present if **MALLOC_SINGLE_POOL** is 0.

If a **malloc** request is denied, i.e., the function returns **NULL**, the requesting process may want to suspend itself awaiting a memory release event. The following function can be used for this purpose:

```
void waitmem ([int pool, ]word state)
```

Its first argument identifies the memory pool (as for **malloc**, it is absent if multiple pools are disabled), and the second one specifies the state where the process wants to be resumed when some memory is returned to the pool. The following macro provides a shortcut for the standard "user" pool:

```
#define umwait(s) waitmem ([0, ]s)
```

If PicOS has been configured with **MALLOC_STATS** set to 1, the following two additional functions become available:

```
word memfree ([int pool, ]address faults)
word maxfree ([int pool, ]address nchunks)
```

where the first argument (in both cases and if applicable) identifies a memory pool. The first function returns the total amount of memory (in words) available for allocation. If the second argument of **memfree** is not **NULL**, it is interpreted as the address at which the function will store the accumulated number of allocation failures, i.e., situations where **malloc** (for the indicated pool) has returned **NULL** because no memory was available. Note that although memory may appear to be available (based on the value returned by **memfree**), **malloc** may still fail for a much smaller requested amount, because the available memory happens to be fragmented. The second function returns the maximum size (in words) of a single memory chunk available for allocation. If **nchunks** is not

NULL, the function will store at the indicated location the total number of chunks (which can be viewed as a measure of fragmentation).

If PicOS has been configured with **MALLOC_SAFE** set to 1, it will perform some rudimentary consistency tests aimed at catching multiple deallocations of the same block, or allocations of blocks not on the free list. This comes at the cost of a slightly increased processing time.

11 Power conservation tools

As of version 3.0, most of the mess related to power conservation from the previous versions has been eliminated, at least for MSP430. The necessary prerequisite for access to deep power saving modes is that the primary crystal attached to the CPU be a low-speed (watch) crystal at 32768 Hz. The high-speed-crystal option (when **HIGH_CRYSTAL_RATE** is 1) makes it difficult to do any non-trivial power savings at all; thus, this requirement isn't really a limitation.

For MSP430, if the primary crystal is in fact low-speed, the situation is very straightforward. PicOS is then compiled (by default) with the so-called **TRIPLE_CLOCK** option, which provides for an efficient implementation of system clocks. The praxis has access to the following operations:

```
void powerdown (void)
void powerup (void)
```

> These functions switch the system between the full-power and low-power modes. By default, and also after **powerup**, the system operates in the full-power mode. A call to **powerdown** selects the deepest sleep state during the periods of CPU inactivity that still makes it possible for the CPU to wake up on events. For MSP430, **powerdown** selects the LPM3 sleep state, where practically all the CPU components are switched off except for the basic clock (ACLK) driven by the primary crystal. If there is a secondary crystal (XTL2) connected to the chip, that crystal is disabled. External interrupts (e.g., from the RF module) are allowed to occur; however, the wakeup time (and transition to full readiness) may be slower with respect to the normal **powerup** mode (in which the sleep state is the most shallow).

While in **powerdown** mode, the praxis can basically function in the same way as in **powerup**, except that the event (including timer) wakeup time may be slightly increased. All system clocks will function normally.

If the primary CPU crystal is high-speed, **TRIPLE_CLOCK** is forced off and then **powerdown** will bring little reduction in power consumption, although it still selects the LPM3 mode for inactive CPU. In that case, the implementation of system clocks is also less efficient.

As a legacy option (and the only option available on eCOG at the moment), **TRIPLE_CLOCK** can be disabled at compilation (also when the primary crystal is low-speed). If **TRIPLE_CLOCK** is 0 and the primary crystal is low-speed, then two more operations become available to the praxis:

```
void clockdown (void)
void clockup (void)
```

These functions change the rate of clock interrupts, i.e., the frequency of internal clock ticks. By default (and after **clockup**), the clock runs 1024 times per second. This allows the system to measure time with the granularity of slightly below 1 millisecond, and this is also the resolution of **delay** intervals. After **clockdown**, the clock slows down to 1 tick per second. Time measurement (in seconds) is still accurate, and **delay** operations work, except that they are rounded to the much coarser grain.

Note that, although **clockdown** and **clockup** are formally available in all circumstances, they are only non-void when 1) the primary CPU crystal is low-speed (at 32768 Hz), **and** 2) the **TRIPLE_CLOCK** option has been **explicitly** disabled at compilation. In all the remaining circumstances, the operations expand into empty statements.

Also note that **clockdown** (when the operation is not void) is automatically forced by **powerdown** (and **clockup** is forced by **powerup**). This is to make sure that when the CPU operates in low-power mode (and significant power savings are in fact possible), the clock is also slowed down to make sure that the CPU does not have to respond to too many events per time unit. This is the consequence of the different implementation of system clocks (with **TRIPLE_CLOCK** disabled), which requires constant interrupts from the timer (normally occurring every millisecond).

This (legacy) mode of operation is messy and discouraged. One more legacy function available in that mode (and also discouraged) is:

```
void freeze (word nsec)
```

This operation freezes (hibernates) the system, with all devices disabled and at the absolutely minimum expense of power for the prescribed number of seconds. The function does not return until the specified number of seconds has elapsed. When it returns, the system continues as if the hibernation period has been completely removed from its lifetime. In particular, the second clock is not advanced during the hibernation.

The freeze operation is only available if the **GLACIER** system option is set to 1 (in addition to **TRIPLE_CLOCK** having been set to 0).

Note that normally, when **TRIPLE_CLOCK** is 1, there is no more need for **freeze**. Executing long delays (page 20) or holds (page 33), with no activities in the praxis, amounts to the same thing (except that the seconds clock is advanced normally).

12 Watchdog

On MSP430, the praxis has access to simple watchdog operations listed below:

```
void watchdog_start ();
void watchdog_stop ()
void watchdog_clear ();
```

Normally (by default) the watchdog is disabled. Once the watchdog is started (after invoking **watchdog_start**) and until it is stopped (with **watchdog_stop**), the praxis will have to make sure to call **watchdog_clear** at least once every second. If it fails to do so, the device will be automatically reset.

13 Selected library functions

In addition to the functions built into the kernel, some useful operations are available in the library (directory *PICOS/Libs/Lib*). They can be referenced after including the respective header files (*.h*). Note that the library directory is automatically searched for praxis includes. Here we only mention a few of those functions. The reader is encouraged to look into *Lib* and glance through the header files for more information.

```
char *form (char *buf, const char *fmt, ...)
```

Requires *form.h*. This function uses the specified format string (**fmt**) to interpret its remaining arguments and encode them into the buffer **buf**. If **buf** is **NULL**, the buffer will be allocated dynamically. In both cases, its pointer is returned via the function value. The set of special codes within the format includes:

- **%d** (signed integer)
- **%u** (unsigned integer)
- **%x** (hex 16-bit word)
- **%s** (string)
- **%c** (character)

Additionally, if the system option **CODE_LONG_INTS** is set to 1, **'d'**, **'u'**, and **'x'** can be preceded by **'l'** to indicate a long (32-bit) operand. No field size indicators are possible.

```
int scan (const char *buf, const char *fmt, ...)
```

Requires *form.h*. The function scans the string in **buf** according to the specified format and assigns extracted values to the remaining arguments (which should be pointers to the properly-sized objects and whose expected number is determined by the number of special fields in the format). The special fields are interpreted as follows:

- **%d** locate the next (possibly signed) integer in the source string
- **%u** locate the next unsigned integer
- **%x** locate the next hexadecimal number
- **%s** extract the next string starting from the first non-white-space character and terminating on the first white space or comma, swallowing the comma if it occurs
- **%c** extract the next character.

The "current" pointer to the source string is updated after each extraction. If the **CODE_LONG_INTS** option is set, the **'l'** (for long) prefix is applicable to the numerical descriptors, e.g., **'%ld'**. The function returns the number of items that have been located and decoded from the input string.

```
bool isdigit (char c)
bool isspace (char c)
bool isxdigit (char c)
char hexcode (char c)
```

These are macros defined in *form.h*. The first three are Boolean operators, with **isspace** returning 1 on any white space character (blank, NL, CR, tab) and 0 on

any other byte. The last macro, **hexcode**, returns the numerical code of a hexadecimal digit.

int ser_select (int port)

This function requires *ser.h*. It selects the UART to which the input/output operations **ser_out**, **ser_outf**, **ser_in** and **ser_inf** (described below) will be applied. By default, this is UART A. The argument can be 0 for UART A or 1 for UART B. The function returns the previous UART selection (0 or 1). When only one UART is configured into the system, the function has no effect.

int ser_out (word state, const char *msg)

Requires *ser.h*. The function writes the specified message to the currently selected UART. Normally, it starts a helper process to send the message asynchronously and returns immediately with the value of zero. To avoid resource problems, the helper process spawned by **ser_out** can only exist in at most one copy (and can only handle one outstanding request). Thus, if the process is already running (still servicing a previous request) when the function is called, the new request cannot be accepted immediately. In such a case, if **state** is not **NONE**, the function marks the current process to be awakened in the specified state as soon as the helper process terminates (i.e., the previous request is completed), and leaves the current process (executes **release**). If **state** is **NONE**, the function returns immediately even if the request cannot be accepted. In such a case, the function returns the *pid* of the currently active helper process. The caller may choose to wait for the termination (**join**) of that process before reissuing the request.

If the first byte of the message is nonzero, it is assumed to be an ASCII string terminated with a **NULL** byte. Otherwise, the message is interpreted as a binary string with the second byte specifying its length. Additionally, the message ends with byte 0x04 viewed as a sentinel. The length passed in the second byte refers to the proper message excluding the three extra bytes; thus, the complete string of bytes looks like this:

[0x00][*length*] ... *length bytes* ... [0x04]

and its total length is ***length*** + 3.

int ser_outf (word state, const char *fmt, ...)

Requires *ser.h*. Acts exactly as **ser_out**, except that there is no binary mode, and the ASCII message may include arguments that will be encoded according to the specified format (as for **form**).

int ser_in (word state, char *buf, int len)

Requires *ser.h*. The function reads a line from the UART and stores it in the specified buffer. The last argument limits the length of the line. Regardless of its value, the line length is hard-limited to 63 characters. If the line is not immediately available, and **state** is not **NONE**, the function blocks the caller, which will be restarted in the indicated state when it makes sense to retry the operation.

If the first character of a new line is nonzero, the line is assumed to be an ASCII string terminated by LF or CR, whichever character comes first. The terminating

character is not stored. A sequence of characters with ASCII codes less than 0x20 (space) occurring at the beginning of a line is ignored. Note that this way lines can be terminated with LF, CR, LF+CR, CR+LF, and multiple consecutive end-of-line sequences are treated as one. If the line starts with a zero byte, it is assumed to be in binary, with the second byte specifying its length, and an extra 0x04 byte appended at the end (see **ser_out**).

The interpretation of the value returned by the function depends on whether **state** is **NONE**. If not, the function can only return when the request has been processed, and then it returns the number of characters stored in the buffer (not counting the **NULL** byte). If **state** is **NONE**, the function returns zero when the request has been processed successfully (the input line was already available when the function was called). Otherwise, similar to **ser_out**, it returns the *pid* of the helper process whose termination event will hint at an opportunity to re-execute the operation.

```
int ser_inf (word state, const char *fmt, ...)
```

Requires *ser.h*. This function performs a formatted read and is a combination of **ser_in** (no binary mode) and **scan**. If **state** is not **NONE**, it returns the number of items that have been located and extracted from the input line. Otherwise, its value is as for **ser_in.**

```
void encrypt (word *buf, int length, const lword *key)
void decrypt (word *buf, int length, const lword *key)
```

Require *encrypt.h*. The first function encrypts the contents of **buf**, **length** words long using the key pointed to by the last argument. The encryption algorithm is TEA. The key is exactly 4 long words (128 bits). When the encrypted buffer is presented to **decrypt** with same key, it will be decrypted to the original plaintext.

```
void hold (word state, lword sec)
```

Requires *hold.h*. The function implements a long delay until the node's seconds clock reaches the specified value. The argument must be global, i.e., its value has to survive the thread's state transitions, as the function may retry at the indicated state before returning (when the specified second has been reached). Here is an illustration how the function should be called:

```
...
static lword del;
...
    del = 3600 + seconds ();

  entry (HOLD_ME)
    hold (HOLD_ME, del);
    delay elapsed
      ...
```

Note that the function returns when the delay has elapsed. Otherwise, it uses the state argument to set up internal alarm clocks (operation **delay**) and monitor the progression of waiting time.

Let us conclude this section with a recommendation for positioning the state argument on the argument list of functions that need it. In PicOS, any function that may have to

block waiting for an event needs a state argument pointing to the place where its calling process should be resumed after the event. We recommend that if the function should be re-executed after the event (i.e., the event signals a new opportunity to try again, like in **ser_out** or **ser_in**), the state argument be the first item on the argument list. On the other hand, if the event signals the completion of whatever the function was supposed to accomplish (like in the system function **delay**), the state argument should be placed at the end.

14 Process tracing and debugging

By including the file *trc.h*. from *PicOS/Libs/Lib* in the header of a praxis file, you will redefine the statements (macros) **process, thread, strand,** and **entry** to write a line of text to UART A whenever the process is activated. This line (produced by calling **diag**) identifies the process and its state.

In addition to **diag**, *sysio.h* defines macros named **dbg_0(v)**,…, **dbg_9(v)**, **dbg_a(v)**,…, **dbg_f(v)**, 16 of them altogether. These macros are intended for rudimentary and compact tracing of programs, possibly during production. Normally, they are defined as empty statements. By setting the system option **dbg_level** to a nonzero value, you enable those macros whose numbers have their bits set in **dbg_level**. For example, is **dbg_level** is 0x80C1, macros **dbg_0**, **dbg_6**, **dbg_7**, **dbg_f** become enabled.

15 Internals

This section 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.

15.1 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

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[13] 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 any) to go first. The initialization sequence ends by entering the *scheduler loop*.

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

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 was described in section 3. 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, multithreaded (safely 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.[14] 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 15.3).

- 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 *code method*).

- 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.

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

- 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.

15.2 Processes and events

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

```
typedef struct    {
      word        Status;
      word        Timer;      /* Timer wakeup tick */
      code_t      code;       /* Code method pointer */
      address     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 **code_t** is declared (in **sysio.h**) as:

```
typedef int (*code_t)(word, address);
```

and represents a pointer to the C function containing the process code method. For those processes (strands - see section 3) that use private data, 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** - see page 20). 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 a process up 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 are awaited by the process, if any, are represented in the **Events** array whose single element looks like this:

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

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

The Type field of an active entry is never zero: a zero value means that the entry is not used. For a **when** (or, equivalently, **wait** - page 19) request issued from the praxis, Type is 1. Here is the list of all possible values of that field:

```
#define ETYPE_USER      1     /* User signal */
#define ETYPE_SYSTEM    2     /* System signal */
#define ETYPE_IO        3     /* I/O */
#define ETYPE_TERM      4     /* Process termination */
#define ETYPE_TERMANY   5     /* joinall */
```

This modest proliferation of event types results from the fact that the system uses internally some events that are not directly available to the praxis. Those events must be told apart from **when**-type events seen by the praxis; thus, the event number alone might not do. Notably, **ETYPE_SYSTEM** is obsolete (not used any more), and **ETYPE_IO** (used solely by the old UART driver) is on its way out. **ETYPE_TERM** and **ETYPE_TERMANY** are used by operations **join**, **joinall**, and **kill** (page 20) to represent process termination events.

Whenever a process issues a **when** request, the system takes the first free entry from the **Events** array in the process's PCB, fills it with the parameters of the request (event number, state) using 1 (**ETYPE_USER**) for the Type field. Then it increments the NE field in the PCB's **Status** word.

The pool of processes known to the system is described in an array of PCBs 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:

```
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 **fork** (page 17) is the PCB address of the created process cast to type **int**. When a process is run, the system variable **zz_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 PCBs. 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].Status & 0xfff0)
#define wakeuptm(p)     (p)->Status &= 0xfff0
#define cltmwait(p)     (p)->Status &= 0xfff7
```

Here are three macros operating on events (the argument points to an **event_t** structure):

```
#define efree(e)           ((e).Status == 0)
#define setestatus(e,t,s)  ((e).Status = ((s) << 4) | (t))
#define getetype(e)        ((e).Status & 0xf)
```

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 simpler (macro **wakeuptm**). 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 **zz_uwait** in **kernel.c**):

```
void zzz_uwait (word event, word state) {

      int j = nevents (zz_curr);

      if (j == MAX_EVENTS_PER_TASK)
            syserror (ENEVENTS, "wa");

      setestatus (zz_curr->Events [j], ETYPE_USER, state);
      zz_curr->Events [j] . Event = event;
      incwait (zz_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** (introduced conceptually on page 19), which delivers a **when** event:

```
int zzz_utrigger (word event) {

  int j, c;
  pcb_t *i;

  c = 0;
  for_all_tasks (i) {
    if (i->code == NULL || nevents (i) == 0)
      continue;
    for (j = 0; j < nevents (i); j++) {
      if (i->Events [j] . Event == event &&
                  getetype (i->Events [j]) == ETYPE_USER) {
        /* Wake up */
        wakeupev (i, j);
        c++;
        break;
      }
    }
  }
  return c;
}
```

The function simply scans through all PCBs, and for every process (nonempty slot) examines its list of awaited events looking for one matching the specified argument. If a matching entry is found, the process is awakened. While at first sight the operation looks inefficient: one obvious way to fix it would be to have lists of processes awaiting specific events, such lists would cost precious memory and, moreover, would likely bring about its fragmentation. On the other hand, if the number of processes is not excessively big, and their lists of events are not excessively long, the overhead is negligible, while no additional memory is required to implement the fundamental concept of awaiting and signaling events.

With the default setting of **MAX_EVENTS_PER_TASK == 4** and **MAX_TASKS == 16**, a single PCB takes 24 bytes, and the total amount of memory used to represent processes is 384 bytes. Both constants can be reduced: **MAX_EVENTS_PER_TASK == 3** should be OK for most praxes (yielding 20 bytes per PCB), while many praxes run no more than 4 or 5 processes at a time.

15.3 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 a fixed-size array of PCBs. When a new process is forked, the system will put its PCB into the first free slot looking from the beginning (as implied by **for_all_tasks** - page 38). 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 (zz_curr) {
	  if (zz_curr->code != NULL && !waiting (zz_curr)) {
	     (zz_curr->code) (tstate (zz_curr), zz_curr->data);
	        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. 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
	      zz_curr = (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 **zz_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** (page 6). 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 zz_restart_entry\n"\
        "zz_restart_entry: mov %0, r1"\
                :: "i"(STACK_START): "r1")

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

SET_RELEASE_POINT declares a global entry point (named **zz_restart_entry**) which basically 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 15.4.

The rest of the scheduler loop is extremely straightforward: it locates the first ready process (the PCB entry is not free and the process is not waiting for anything) and calls its code method passing it the state number and data pointer as the arguments. Whenever that happens (i.e., a ready process is found and its code method is called and it subsequently returns), the scheduler starts from the beginning.[15] 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 (zz_systat.pdmode) { \
                  cli; \
                  if (zz_systat.evntpn) { \
                        sti; \
                  } else { \
                        _BIS_SR (LPM3_bits + GIE); \
```

[15] Note that when the code method executes **release**, the scheduler loop will be automatically entered from the beginning. 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.

```
                } \
            } else { \
                cli; \
                if (zz_systat.evntpn) { \
                    sti; \
                } else { \
                    _BIS_SR (LPM0_bits + GIE); \
                } \
            } \
            zz_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 **zz_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 **zz_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 zz_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 **zz_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 stored 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 (see section 11) 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 **zz_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) runnable. Note that the reason why the sleep state has been entered in the first place was that no process was found ready to run at some point. 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:

```
entry (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.,

```
entry (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 **zzz_utrigger** (page 39). 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 zz_systat.evntpn = 1
```

and this one upon return:

```
#define RTNI do { \
                  if (zz_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 method). 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).

15.4 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 (page 22), blinking LEDs, low-level debouncers for I/O pins, etc.

3. The delay clock. This clock is used to implement the **delay** operation (page 20) available to praxis processes.

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 **zz_old**, **zz_new**, and **zz_mintk**. For aesthetic reasons, we shall strip the prefix **zz_** 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 routine 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 40). 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 (zz_old, zz_new, zz_mintk)) {
    for_all_tasks (i) {
      if (!twaiting (i))
        continue;
```


```
        if (twakecnd (zz_old, zz_new, i->Timer)) {
          wakeuptm (i);
        } else {
          d = i->Timer - zz_new;
          if (d < min)
            min = d;
        }
      }
    } else {
      if (zz_mintk - zz_new < min)
        goto MOK;
    }
    zz_mintk = zz_new + min;
MOK:
    zz_old = zz_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 those 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 38) 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 (zz_curr, state);
      cltmwait (zz_curr);
      update_n_wake (d);
      zz_curr->Timer = zz_old + d;
      inctimer (zz_curr);
}
```

See page 38 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 eCOG-compatibility of the system. With Cyan SDK for the eCOG, 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 (zz_systat.ledsts) {
      if (zz_systat.ledblc++ == 0) {
            if (zz_systat.ledblk) {
                  if (zz_systat.ledsts & 0x1)
                        LED0_ON;
                  if (zz_systat.ledsts & 0x2)
                        LED1_ON;
                  if (zz_systat.ledsts & 0x4)
                        LED2_ON;
                  if (zz_systat.ledsts & 0x8)
                        LED3_ON;
                  zz_systat.ledblk = 0;
            } else {
                  if (zz_systat.ledsts & 0x1)
                        LED0_OFF;
                  if (zz_systat.ledsts & 0x2)
                        LED1_OFF;
                  if (zz_systat.ledsts & 0x4)
                        LED2_OFF;
                  if (zz_systat.ledsts & 0x8)
                        LED3_OFF;
                  zz_systat.ledblk = 1;
            }
            if (zz_systat.fstblk)
                  zz_systat.ledblc = 200;
      }
      TCI_MARK_AUXILIARY_TIMER_ACTIVE;
}
```


The **ledsts** attribute of **zz_systat** (see page 42) 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 52).

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 (page 20). This clock can be implemented by a dedicated a timer interrupt occurring every second and adding 1 to the global variable **zz_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* **zz_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 (see section 11) 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.[16] 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 (zz_utims [0] == 0)
                  goto EUT;
            if (*(zz_utims [0])) {
                  (*(zz_utims [0]))--;
                  aux_timer_inactive = 0;
            }
            ...
```

[16] 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.

```
EUT:
#include "irq_timer.h"

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

      // The seconds clock
      TCI_CCS += TCI_SEC_DIV;
      zz_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 49). 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[17] 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;    // if 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**):

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

```
static word gettav () {
        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 (**zz_nseconds**) is then advanced. Following that, any snippets brought in by **second.h** (see page 50) 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;
        zz_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 **zz_new** (see page 45), 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 46) is to execute **TCI_UPDATE_DELAY_TICKS** in order to make sure that NEW (**zz_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**), **zz_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 48). 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) {
            zz_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,[18] i.e., NEW (**zz_new**) is up to date the way it looks.[19] 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 = zz_mintk - zz_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 (**zz_mintk**) and OLD (**zz_old**) - see page 45. 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., multiplied by 4).

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

[19] Note that there is no need to worry about the validity of NEW is 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.

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 eCOG), 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-op.

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

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

which should be inserted into the sanitized version shown on page 46 right after the label **`MOK`**. Variable **`millisec`** accumulates ticks towards a second (**`JIFFIES`** is 1024) and **`zz_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.