Olsonet Communications

Pic OS

Version 1.0
September 22, 2003

Preamble

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

PicOS Virtual Transceiver Interface is a sister document, referred to as [TCV].

1. Introduction

The primary problem with implementing classical multitasking on microcontrollers with limited RAM is minimizing the amount of fixed memory resources that must be allocated to a process. One example of such a resource is the stack space. Namely, to make sense out of this idea, every task must receive a disjoint 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 eCOG,[1] 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 area* 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.

Consequently, the multitasking adopted by PicOS is not *classical*. The different *tasks* share the same global stack and act as coroutines with multiple entry points and implicit control transfer. Each 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 (at least to some extent), which is especially useful and natural for reactive applications, i.e., ones that are not CPU bound, but perform many finely-grained *tasks* in response to possibly complicated configurations (sequences) of events.

The programming paradigm of PicOS stems from SMURPH (also called SIDE in its most recent version), which was intended as a programming language for describing reactive

[1] eCOG1 is a microcontroller from Cyan (www.cyantechnology.com). We use it as an experimental platform for our works.

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 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, instead of their models. PicOS can be viewed as a verification of this idea in the microcontroller domain.

2. Processes

By a popular tradition, tasks in PicOS are called *processes*. A process 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 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 application and is responsible for creating all other processes. Some processes, typically device drivers, may be started internally by the system before the root process.

The multiple entry points of a process are identified by integer numbers from 0 to 4095. Whenever a process is run (receives 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 function is executed at state 0. Thus, state 0 can be viewed as an initial state.

When it is run, a process may issue (explicit or implicit) *wait requests* identifying conditions (e*vents*) to resume it in the future, and associating states with those events. At some point the process puts itself to sleep (when this happens, we say that the process has completed its current state). It 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 wait request.

Note that a sleeping process may be waiting for more than one event. 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 wait requests were issued is immaterial. All the wait requests executed by the process in a given state aggregate to define an unordered collection of awaited events.

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

If a process goes to sleep without issuing a single wait 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 explicitly 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.

Owing to the fact that a process in PicOS needs 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, 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 = 5+2E words

where *E* is the maximum number of events. With the default value of *E=4* (which so far has proved sufficient for all our applications), this formula yields 13 words (or 26 bytes).

3. System organization

The present version of PicOS executes on the eCOG evaluation board and comes with a few sample applications whose purpose is to test the system and illustrate its principles. The system is configured with five devices (having their drivers present in the system): the two DUARTs (handled by the same driver), the LCD display, the four LEDs (viewed as a simple output device capable of showing the contents of one 4-bit nibble), the Ethernet interface, and a wireless channel (in a few configurable flavors).

In addition to this collection of devices, the system is equipped with a generic transceiver (TCV) module implementing open-ended session management intended for networked applications. The transceiver module can be associated with physical interfaces, and its functionality (as perceived by the application) is extensible by *plugins*.

For example, within the framework of the evaluation board, the transceiver module can be associated with the wireless channel or with the Ethernet interface in a way that emulates a radio channel possibly shared by multiple boards interconnected via a local network. Such a farm of evaluation boards, additionally equipped with an emulation server (e.g., running on a Linux workstation), can be used as a powerful testbed for developing wireless applications on eCOG-controlled hardware.

It is also possible to use the Ethernet interface as a straightforward network connection. In combination with TCV, which turns this connection into a plugin-controlled flexible networking device, this option makes it possible to build embedded networked applications operating directly on the evaluation board. In particular, the TCP/IP stack can be implemented as a plugin to the TCV module.

The directory tree of the system is called `Platform`, and it consists of the following sub-directories:

- `PicOS`

  This directory contains the source code of the system kernel, all device drivers, and the TCV module.

- `Lib`

  This directory contains a few handy programs (in C) that can be referenced by applications. This collection, which includes a sample TCV plugin and three physical interface modules for TCV, has only just started and is likely to grow as more useful examples of typical functions are programmed and added to the library.

- `Apps`

  This is a repository for sample applications with each application occupying one sub-directory. Again, the repository will grow as more applications are developed and integrated with PicOS.

- `Linux`

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

- `Docs`

  Documentation directory.

These are the most relevant files of the kernel (directory `PicOS`):

`main.c and kernel.c` — the proper kernel including the system-level initialization code, the CPU scheduler, and the device drivers, except for the Ethernet and radio devices

`tcv.c` — the TCV module (excluding the physical interface modules and plugins, which belong to `Lib`)

`ethernet.c` — the Ethernet driver

`radio.c`	the radio (wireless) driver
`rasm.asm`	the assembler language helper for the radio driver
`kernel.h`	the basic include file for the kernel
`tcv.h`	the include file of the TCV module
`ethernet.h`	the include file of the Ethernet driver
`tcvphys.h`	the physical layer interface to the TCV module (to be included by library implementations of physical interfaces)
`tcvplug.h`	the plugin interface to the TCV module (to be included by TCV plugins)
`sysio.h`	the standard header file to be included by the application. This file also lists system configuration parameters selecting device drivers, options, and some numerical values. The most relevant of those parameters are settable on a per-application basis in a file named `options.sys`, which should appear in the application directory.

4. Quick start

The eCOG software (version 2 is needed by the present release of PicOS) works smoothly under Cygwin (using GNU *make*), as well as under Visual C++ with *nmake*. In the subsequent instructions, we are assuming the Cygwin platform, although the same results can be easily achieved under other environments (e.g. *nmake* and *app.mak* can be used instead of *make* and *Makefile*).

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

2. Move to `Platform/Apps/Toy`. This directory contains a toy application, which illustrates the operation of three devices: the serial port, the LCD display, and the LEDs. Execute make in that directory. This will create a number of standard files including the loadable image of PicOS integrated with the toy application.

3. Load the *Image* file into the board following the standard procedure outlined in eCOG documentation, and start the program. You should see in the HyperTerminal window a text similar to this:

```
PicOS v0.96, Copyright (C) Olsonet Communications, 2002, 2003
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 viewed as a message to be displayed on the LCD. If that string is not longer than 32 characters, 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.

Example

The way PicOS processes are constructed is illustrated by the following sample process code. It is a slightly simplified version of `Platform/Lib/__outserial.c` which file is (implicitly) referenced by the toy application.

```
#define  OM_INIT   00
#define  OM_WRITE  10
#define  OM_RETRY  20

process (__outserial, const char)

      static const char *ptr;
      static int len;

   entry (OM_INIT)

      ptr = data;

   entry (OM_WRITE)

      if ((len = strlen (ptr)) == 0) {
         ufree (data);
         finish;
```

```
        }
    entry (OM_RETRY)
        ptr += io (OM_RETRY, THE_UART, WRITE, (char*)ptr, len);
        proceed (OM_WRITE);
    endprocess (1)
```

The first parameter of the `process` statement assigns a name to the process code function (e.g., to be referenced when the process is created, see below). The second parameter describes the type of the process's data object, or rather the type pointed to by the data object, which is always represented by an implicitly declared pointer named `data`. As the role of the `__outserial` process is to write a character message to `THE_UART` (i.e., `UART_A`), its data object is a string of characters (i.e., `data` is a character pointer).

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 the process will not be able to start. In our example, `__outserial` commences its execution in state `0` (labeled `OM_INIT`).

The process code function resembles a regular C function and may include declarations of local variables. Note 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, but they look nicer (and safer) when declared in a block following the respective `entry` statement. 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 `__outserial` never exists in more than one copy at a time, the second solution is a natural choice in this case.

The fact that `__outserial` exists in at most one copy at a time is indicated by the parameter of the `endprocess` statement which closes a process definition. The number specified as its argument imposes a limit on the number of copies of the process that can run concurrently. The value of zero stands for no explicit limit.

When started (see below), the process is passed a string of characters to be written to the UART. In fact, the process is not intended to be directly visible by the application (this is why its name is obfuscated with the two underscore characters). The following function (simplified for clarity) provides a user interface to `__outserial`:

```
void ser_out (word st, const char *m) {
```

```
    int prcs;
    char *buf;

    if ((prcs = running (__outserial)) != 0) {
       join (prcs, st);
       release;
    }

    buf = (char*) umalloc (strlen (m) + 1);
    strcpy (buf, m);

    fork (__outserial, buf);
}
```

The function can complete in one of two possible ways: either by simply returning to the caller, or by blocking the caller and later resuming it in a specific state (indicated in the first argument). The `if` statement checks if an instance of `__outserial` is currently running. If it is, the caller has to wait for its completion. This approach serializes the write operations to the UART and avoids messing up the output with multiple (possibly interleaved) line fragments written at about the same time. Additionally, it limits the degree of multiprogramming (and makes it more predictable), which is not irrelevant in a tight resource environment. If another copy of `__outserial` is currently active, the `join` operation declares that the caller of `ser_out` wants to be notified about its termination, while the subsequent `release` effectively blocks the caller (and never returns). The caller will be resumed in the state specified as the second argument of `join` when the active instance of `__outserial` executes `finish`. If the path is clear, the function allocates a buffer to accommodate the string specified by the caller and starts an instance of `__outserial` (operation `fork`) passing it the buffer as its data structure. The process operates independently of the caller and therefore it uses a copy of the original string, which the caller is free to modify or destroy.

Returning to the process code, we can see that in its first state `__outserial` initializes `ptr`, i.e., the pointer to the first unprocessed location in the output string, and falls through to the next state labeled `OM_WRITE`. Then, if `ptr` points to the end of the string, it means that the entire string has been written, in which case the process deallocates the buffer and terminates itself (there is no return from `finish`). Otherwise, in state `OM_RETRY`, it calls `io` to try to write the entire remaining portion of the string to the UART, increments `ptr` by the length of the written portion, and continues at state `OM_WRITE` for as long as some characters still remain to be written. The semantics of `io` is explained below in detail. Its first argument specifies the state to be used when the device is busy. In such a case, the calling process is suspended and it will be resumed in the indicated state when the device becomes available.

The following code fragment illustrates how `ser_out` can be called. It comes from the toy application (file `app.c` in directory `Platform/Apps/Toy`).

```
...
entry (RS_RCMD-1)
```

```
   ser_out (RS_RCMD-1, "\r\n"
      "Welcome to PicOS!\r\n"
      "Commands:  'm...anytext...' (LCD)\r\n"
      "           'b...hexdigits...' (LEDs)\r\n"
      "           's bufsize nkwords' (SDRAM test)\r\n"
      "           'h' (HALT)\r\n"
      "           'r' (RESET)\r\n");

entry (RS_RCMD)
...
```

The function requires a *state* in the calling process to make it possible for the caller to wait for the availability of the output channel (meaning the disappearance of a previous instance of `__outserial`). When the function returns (and falls through to the next state in the above sequence), the character string has been accepted for output.

It may be worthwhile to recommend at this point a convention for positioning the *state* argument on the argument list of functions. 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), 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, the *state* argument should be positioned at the end.

5. Standard types

As exemplified by the sample applications, the user-visible types, functions, and constants provided and understood by the system are defined in file `sysio.h`, which should be included by any application to be integrated with PicOS. In addition to the standard types defined by C, that file 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`

Please consult `sysio.h` for more useful constants and macros, as well as for the global configuration parameters of the system. Specific configuration parameters, settable on a per-application basis, are stored in file `options.sys` in the application directory.

6. Operations provided by the kernel

PicOS is simple enough to be studied and understood by examining the code of its sample applications. Below, we list 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. The new process executes concurrently with its creator. The first argument refers to the process code function (the first parameter of `process`), the second represents the pointer to the process data object. Note that this pointer is a 16-bit `word` pointer. Care should be taken when passing character pointers as data objects. Because of the special way in which such pointers are treated by the eCOG, 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 by `endprocess` – see above), `fork` returns zero and its action is void. A legitimate process Id can never be 0.

```
---> 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`. 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 process Id), such that it still can be perceived as active by `running`.[2] The only way for a zombie to disappear completely is to be killed by some other process. None of these operations: `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. It is legal for `killall` to have no effect (returning zero), if no process is currently running the specified code function.

```
---> int prioritize (int pid, int pr)
```

This function assigns a scheduling priority to the indicated process. As a special case, if the process id is passed as 0 the current (invoking) process is assumed. Otherwise, if the process id is valid and the priority value is within the legitimate range (from 0 inclusive to the maximum number of tasks times 10 exclusive), the priority is updated for the process. Otherwise, if the passed priority value is not legitimate it is not set. In any case, the function returns the process priority. For example, a value of -1 can be passed as the priority argument to get the current priority value.

[2] The zombie state is useful for avoiding race conditions in some IPC scenarios.

```
---> int prioritizeall (code_t code, int pr)
```

This function assigns a scheduling priority to all process instances of the indicated code function and returns the number of processes whose priority value has been changed. If no processes of the indicated code function are found, the function returns 0. The priority value must be within the legitimate range (from 0 inclusive to the maximum number of tasks times 10 exclusive).

```
---> int running (code_t code)
```

If an instance of a process using the specified code function is currently active, `running` returns the process Id of one (any) such instance. Otherwise, `running` returns zero. A zombie is considered active by this function.

```
---> int zombie (code_t code)
```

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

```
---> 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. It provides a way to tell whether a process is a zombie, e.g., `sysio.h` defines this macro:

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

```
---> void wait (word event, word state)
```

Issues a wait request for the indicated event. When the event is triggered, the process will be resumed in the specified state. Although the `state` is formally 16 bits long, its value is truncated to the least significant 12 bits. Thus, a state number must be between 0 and 4095 inclusively.

For the purpose of `wait` and `trigger` (see below), events are identified by 16-bit numbers viewed as bit patterns without any implied semantics. Certain (special purpose) events can be awaited with some other operations (e.g., `join`, `io`, `delay`). Such events are never confused with events awaited by `wait` and triggered by `trigger`.

```
---> int join (int pid, word state)
```

Issues a wait request for an event to be triggered when the indicated process terminates itself (or becomes a zombie). No wait request is issued if the process already is a zombie, or if it doesn't exist. The function returns the argument `pid` or zero if the process doesn't exist.

```
---> int joinall (code_t code, word state)
```

Issues a wait request to be triggered when any process running the specified code function terminates itself 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 zombies already.

```
---> int trigger (word event)
```

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

```
---> void delay (word timeout, word state)
```

Issues a timer wait request, i.e., sets up an alarm clock to go off after the specified number of *milliseconds*.[3] When that happens, the process will be restarted in the indicated state. At most one alarm clock per process can be set 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 `wait` request and can coexist with (other) `wait` requests (with the obvious semantics).

```
---> void snooze (word state)
```

This operation makes it possible to continue an interrupted `delay`. Suppose that a process issues a wait request for some event together with a `delay` request. If the event occurs before the timer goes off, the process may want to handle the event and then continue waiting for the timer (for whatever time is left to its expiration). `snooze` has the same semantics as `delay` with the first argument equal to the leftover time from the last call to `delay`. This interpretation is valid for as long as the timer's original delay does not expire. Executing `snooze` after that time will result in a random delay, which can be anywhere within the original specification at the last `delay` call.

```
---> 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.

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

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

This operation is exactly equivalent to the following sequence:

```
join (fork (code, data), state);
release;
```

and is implemented as a macro. It creates a new process and puts the parent process to sleep until the child terminates itself. When this happens, the parent is resumed in the indicated state.

---> 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_A, UART_B, LCD, LEDS, ETHERNET, RADIO, and the operations (op) are READ, WRITE, and CONTROL.

The five arguments stand for the blocked state identifier, the device number, the operation (another integer number), the buffer pointer, and the buffer length. In general, 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 which case, io returns the number of those bytes. Otherwise, the device is considered busy and the function *may not return*. In such a case, 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 (NONE is recommended in such a case – see below).

The way io awaits a status change on the device fits the general idea of (possibly multiple) wait 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 wait 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 void `READ/WRITE` request is one specifying zero buffer length, i.e., zero bytes to be transferred. Such a request is always legitimate and it does nothing.

As a significant number of `io` requests 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.

```
---> int utimer (address, bool)
```

This operation declares or removes a countdown timer. The first argument points to an unsigned integer number. If the second argument is `YES`, (nonzero) the `word` location specified in the first argument will be added to the pool of countdown timers. As soon as its contents are set to a nonzero value, 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.[4] For an addition of a new timer, the function returns the new size of the timer pool. For a deletion, it returns the original size of the pool or zero if the specified timer wasn't found in the pool.

Note: If spare SDRAM is used by the application (see below), 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.

```
---> long seconds (void)
```

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

```
---> udelay (word del)
```

[4] The maximum size of the pool is set at 4 in a somewhat rigid and unparameterizeable manner (using an unwrapped loop for efficiency).

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

```
---> mdelay (word del)
```

This is another spin delay loop that takes approximately `del` milliseconds.

```
---> 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 standard operations on strings. They behave almost like the corresponding functions from UNIX except that the copy/cat functions return no values. The remaining differences are:

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.

7. Devices

8.1 UART_A and UART_B

If configured into the system, the two UARTs are available as devices number 0 (symbolic constant `UART_A`) 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 the standard rules (as 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 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. The legal rates are 1200, 2400, 4800, 9600, 19200, and 38400 baud.

The purpose of the locked mode is to make it possible to use the UARTs for emulating naive serial devices (e.g., simple radio interfaces). When put into the locked mode, the device operates in a persistent, immediate, and interrupt-less manner. The following control operation serves this end:

```
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.[5] `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).

[5] 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.

8.2 LCD

The number of this device is 2 (symbolic constant `LCD`). No `READ` operation is available (the LCD is solely an output device), and `WRITE` sends the specified text to the LCD up to the maximum of 32 characters, which is how much the device can accommodate (in two rows of 16 characters each). Any characters written past the 32-character limit are ignored and discarded. Note that `WRITE` can block, as writing to the LCD involves delays.

The device driver maintains a position counter (its value is between 0 and 32) determining where the next written characters will appear on the display. This counter is initialized to zero when the system starts up and then it is incremented by one with every character written. When it reaches 32, the subsequent characters sent to the LCD are ignored. At any moment, the position counter can be explicitly reset to any value between 0 and 31 with the following request:

```
io (state, LCD, CONTROL, &pos, LCD_CNTRL_POS)
```

where `pos` is a single character (whose integer value should be between 0 and 31) indicating the position. In addition, the LCD buffer can be erased with another `CONTROL` operation that looks like this:

```
io (state, LCD, CONTROL, NULL, LCD_CNTRL_ERASE)
```

Due to the asynchronous and timer-driven operation of the LCD driver, both requests can block; thus, the `state` argument is relevant.

8.3 LEDS

This very simple device, numbered 3 (symbolic constant `LEDS`), provides a single never-blocking `CONTROL` operation for setting the four LEDs on the board. The respective `io` request looks like this:

```
ion (LEDS, CONTROL, &c, LEDS_CNTRL_SET)
```

where `c` is a character whose four least significant bits determine the setting of the four LEDs.

8.4 ETHERNET

The Ethernet chip (SMSC 91C111) is available as device number 4. This is the most complicated device in the present version of PicOS, offering several modes and options. By default, the device is set up to operate in a non-promiscuous manner (i.e., ignoring frames addressed elsewhere), but to accept multicast packets.

In this standard (raw) mode, `READ` attempts to receive a complete frame from the interface, while `WRITE` sends a complete frame. In both cases, the `io` operation can block: until a received frame shows up in the chip's FIFO (`READ`), or until enough buffer space is available within the FIFO to accommodate a new outgoing frame (`WRITE`).

In both cases, the buffer argument of `io` refers to a complete frame, including the MAC header (14 bytes) but excluding the (trailing) CRC code. This means that for `WRITE`, the caller is responsible for providing the correct destination address and protocol Id in the frame header. The sender address field, however, will be filled in by the driver with the actual MAC address of the interface (note that the frame provided by the caller must include room for it).

The shortest written frame must consist at least of a complete MAC header, i.e., its minimum legitimate length is 14 bytes. Note that short frames transmitted over Ethernet interfaces are inflated to the minimum of 60 bytes. If the input buffer to accommodate a received frame (the length parameter of `READ`) is shorter than the actual frame, only an initial portion of the frame will be returned to the caller, and the remainder will be irretrievably lost.

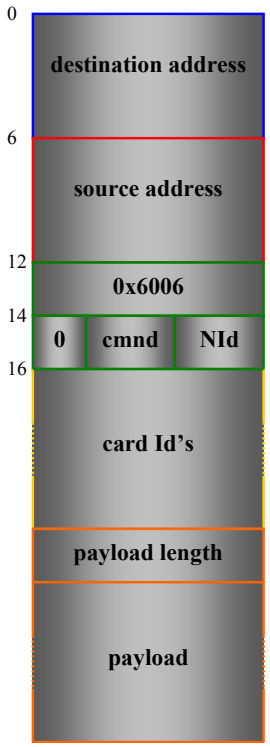

Figure 1. Downlink cooked frame.

In addition to the raw mode, the device offers the so-called *cooked* mode, which assumes a special format of the Ethernet frames. This format is primarily intended for emulating wireless channels, but it can be used for other purposes, e.g., to communicate embedded applications running on (possibly multiple) evaluation boards with a server connected to the same LAN. In the cooked mode, the frame structure becomes transparent to the user, and the `READ/WRITE` operations handle "logical" packets devoid of Ethernet-specific MAC-level framing. The possibly multiple boards connected to the same Ethernet LAN are identified by 16-bit integer values (called card Id's) which can be assigned to them via a `CONTROL` operation addressed to the `ETHERNET` device (see below). Cooked packets are passed as MAC payloads in Ethernet frames, with the packaging being done automatically by the `ETHERNET` driver.

Figure 1 shows the frame format of a downlink cooked packet, where by "downlink" we mean "addressed to a card" (as opposed to a packet being sent by a card and addressed to the server). The first 14 bytes constitute the standard MAC header, with the protocol Id of `0x6006`, which code (not used by any popular protocol) identifies the cooked format. The most significant bit of the next byte (i.e., the first byte of the MAC payload) should be zero to indicate the downlink direction of the packet. The remaining bits of that byte are interpreted as the "command" describing the type of the cooked packet. In the present version of the driver, this field is not interpreted and assumed to be zero (all downlink packets that qualify for reception are just received and their cooked payloads are made readable).

Byte number 15 tells the number of entries in the card Id table starting at byte 16 and consisting of word-sized card identifiers. The purpose of that table is to list the cards to which the packet is addressed. Having received a (possibly multicast) cooked frame, the driver checks if the logical identifier that has been assigned to the board (see below) occurs on the list. The packet is only received (and made available for `READ`-ing) if this is the case. If NId is zero (i.e., the list is empty), the packet is intended for all boards on the network.

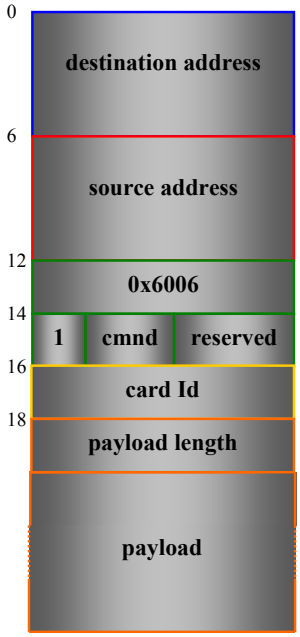

Figure 2. Uplink cooked frame.

The 16-bit payload length field follows the list of card identifiers. It specifies the number of bytes in the payload part, which constitute the packet contents. These are the only bytes of the frame that will be retrieved by `READ`.

The layout of an uplink frame (shown in Figure 2) is a bit simpler, as such a frame is never addressed to a set of recipients, but to a (single) server. Thus, the "command" word is followed by the (single) card Id of the sending board, which in turn is followed by the payload length. The most significant bit of byte 14 is 1 to indicate that this is an uplink packet. Such a packet, if received by the `ETHERNET` driver, will be ignored.

A `WRITE` operation in cooked mode interprets the buffer contents as the payload of an uplink cooked frame. The driver is responsible for building the complete frame, including the source and destination addresses. Usually, the MAC destination address of a cooked frame is unicast and identifies the server. The driver learns that address when it receives the first downlink frame (in which the server address arrives as the source address). If an uplink frame is sent before the first downlink frame has been received, the driver multicasts the frame.

The raw/cooked mode is selectable via `CONTROL` operations on the fly and independently for `READ` and `WRITE`. In addition, for `READ`, it is possible to select a flexible mode, whereby both types of frames (raw and cooked) will be received correctly. To let the caller unambiguously recognize the received frame type, the driver provides a `CONTROL` call returning the mode of the last-received frame.

Below we list the CONTROL requests available for the ETHERNET device. None of those requests ever blocks. To simplify their description, we only mention the symbolic request code, i.e., the value of the fourth (len) parameter of the io operation (the request identifier), and the value passed via the buffer argument of io (the request parameter).

ETHERNET_CNTRL_PROMISC

The request sets or clears the promiscuous mode for the interface. If the first byte of the string pointed to by the request parameter is nonzero, the interface is put into the promiscuous mode whereby it will receive all frames. Otherwise, the promiscuous mode is cleared.

ETHERNET_CNTRL_MULTI

This requests sets or clears the "all multicast" reception mode, in a way similar to ETHERNET_CNTRL_PROMISC. In the "all multicast" mode, the card receives all multicast frames. This mode is implicitly assumed if the READ mode is cooked (or flexible), but it can be set explicitly, e.g., for the raw READ mode as well.

ETHERNET_CNTRL_SETID

The parameter is assumed to point to a 16-bit (unsigned) word (which must be cast to char* in the io call). The request sets (or resets) the logical card Id of the interface.

ETHERNET_CNTRL_RMODE

The first byte of the parameter string is interpreted as an integer designator of the READ mode to be assumed by the device. The legal values are ETHERNET_MODE_RAW (0), for the raw mode, ETHERNET_MODE_COOKED (1), for the cooked mode, and ETHERNET_MODE_BOTH (2) for the flexible mode.

ETHERNET_CNTRL_WMODE

This request sets the WRITE mode, in a way similar to the previous request. This time, however, only modes 0 and 1 are supported (the flexible mode makes no sense in this direction).

ETHERNET_CNTRL_GMODE

The parameter is ignored. This operation returns (via the function value) the mode of the last-read frame, and its primary purpose is to tell the actual type of a frame retrieved in the flexible READ mode. The possible values returned by io are 0 (ETHERNET_MODE_RAW) and 1 (ETHERNET_MODE_COOKED).

ETHERNET_CNTRL_ERROR

The parameter is expected to point to a 16-bit word where the operation will store the last error status of a READ operation.[6] The internal error status is then cleared. The value of zero stands for no error. Except for this error status indication, erroneously received frames are tacitly discarded and ignored. A special value of the error status, 0xffff, indicates a format error in a cooked frame, e.g., length field out of range.

ETHERNET_CNTRL_SENSE

If the "buffer" parameter is NULL or its first byte is zero, the operation checks if the receive FIFO of the interface contains a frame waiting to be read. The function returns 1 if this is the case, and zero otherwise. If the first byte of the parameter is nonzero, the input FIFO is additionally erased (all pending frames are discarded) when this check is made.

All those CONTROL requests that are not expected to return a function value return 1. The general idea is that simple (Boolean) status values are returned directly by the function, while more complex values are returned through the parameter. This is because io cannot return some values, e.g., -1.

8.5 RADIO

The RADIO device interfaces to the system a simple wireless board (three types of such boards are currently supported, and the list is likely to grow) and provides raw read/write access to the wireless channel. The READ operation attempts to receive a packet and always returns (never blocks). If no packet has been received, the value returned by the operation is zero; otherwise, it returns the received packet length in bytes.

This simplistic treatment of the RADIO interface caters to the lowest common denominator of the supported radio boards (both at present and in the future). Many such boards make it impossible (difficult, unnatural, unreliable) to use interrupts for waking up the driver when a frame is ready to be received. Consequently, the application (or the TCV plugin handling the radio interface) may prefer to deal with a simple interrupt-less device using its private means (e.g., application-specific timing) to determine when a frame reception should be attempted. A non-standard event triggering mechanism is provided as a reception option, with the exact semantics of the events depending on the radio board. In some cases, those events are "simulated interrupts" triggered by a timer-based polling of the received signal. In some other cases (e.g., XEMICS 1202) those events are actually interrupts triggered by the pattern recognition feature of the chip (responding to a recognized received frame preamble).

[6] The error status is a collection of four bit flags corresponding to four possible framing errors diagnosed by the chip. It is recommended to interpret this value as a binary error indicator, with zero meaning OK.

The three radio boards currently handled by PicOS are: RFM Virtual Wire DR 1200-DK development board, Valert's experimental board (not generally available), and XEMICS 1202 915MHz board. They are selected by setting `RADIO_TYPE` in `sysio.h` to one of three values: `RADIO_RFMI`, `RADIO_VALERT`, `RADIO_XEMICS`. Note that additionally `RADIO_DRIVER` must be set to 1 (in `options.sys`) to enable the radio device.

The raw packets handled by the `RADIO` device have no headers, except for physical preambles, which are automatically inserted and stripped by the driver. However, they do have trailers in the form of checksums, which are needed to verify the correctness of their reception. The number of bytes in a packet (the buffer length for `READ`) must be a multiple of 4. The last four bytes of a packet are assumed to be the checksum. Those bytes are returned together with the packet, and they count to the specified buffer length.

The minimum buffer length for `READ` (i.e., the minimum packet length) is 8 and it includes the 4 bytes of checksum. If the received packet (together with its checksum) is longer than the specified length, it will not be formally received (the operation will return zero, as if the packet weren't noticed by the receiver).

Note that something potentially sensible may be returned in the buffer, even if the `READ` operation formally fails and returns zero. For example, if a packet is received completely but its checksum is incorrect, the packet will be stored in the buffer, although `READ` will return zero. An incorrect checksum usually implies lost synchronization to the received packet, which means that starting from some (generally unknown) point, the received bits make little sense.

The same restrictions on buffer length also apply to `WRITE`. The buffer must consist of an entire number of 4-byte long-words (at least 2), with the last long-word providing a placeholder for the checksum. Its initial value is irrelevant. Before transmitting the packet, the driver will calculate the checksum and store it in the last four bytes of the buffer. Similar to `READ`, `WRITE` never blocks and blindly transmits the supplied packet.

There is a `CONTROL` request way to switch off checksum control as well as the restrictions on packet length (see below). One should be aware, however, that the radio channel is inherently very noisy and capricious. Thus, if the application decides to run with the standard error control disabled, it should use its private means to ensure reliability of data exchange.

Two bit encoding schemes are currently implemented. The first of them, dubbed *biphase* encoding, is used for `RADIO_RFMI` and `RADIO_VALERT`. It relies on measuring intervals between on/off transitions of the radio signal. The encoded unit (the physical *symbol*) consists of three physical *bits* (signal slots) corresponding to two logical bits. The first physical *bit* of a triplet is set to the reverse of the previous signal level, and the next two *bits* are set to reflect the values of the two logical bits. The binary interpretation of the high/low signal is flipped after every symbol (triplet) to provide for a better signal balance under a long stream of identical logical bits.

Transmitted packets are preceded by preambles. A preamble consists of an alternating sequence of high and low signal, with the high signal duration equal to three physical *bits* and the low signal duration equal to one physical *bit*. This imbalance is intended to make the high signal level more perceptible, e.g., to selective polling. The preamble ends with a high signal lasting for two physical *bits*, which is followed by the first symbol of the proper (receivable) packet. The packet ends at a low signal lasting for more than three physical *bits*. The actual number of (logical) bits in the packet is determined from the requirement that it must be a multiple of 32.

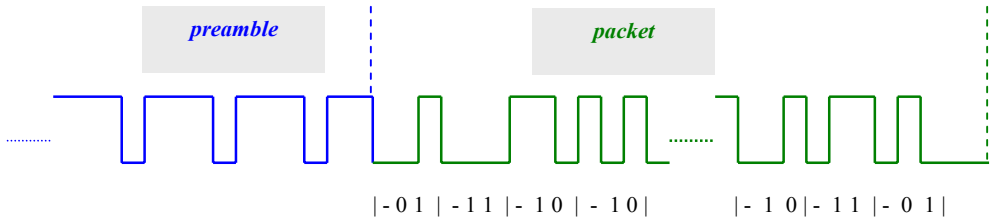

Figure 3. Sample packet encoding with the biphase scheme.

Figure 3 shows a sample packet encoding. Note that there is a signal transition on every symbol boundary, and, consequently, the longest legitimate duration of an unchanged signal is three physical *bits*. For decoding, the first physical *bit* of every symbol is ignored, and the interpretation of 0 and 1 in the remaining *bits* reverses with every new symbol.

Ignoring the preamble and other overheads (checksums, headers, collisions, errors), the effective bit rate is equal to 2*b*/3 bits per second, where *b* is the raw rate expressed in physical *bits* per second. This raw rate is settable by the application within some limits (see below).

The second encoding scheme used by the XEMICS board is more straightforward and simply consists in interpreting the high signal level as 1 and the low level as 0. This is possible because the XEMICS chip processes the received signal internally and makes it available along with the data clock signal that strobes the received bits. To make it work, the selection of available bit rates is restricted to the standard values: 4800, 9600, 19200, 38400 bps. Additionally, at least one bit transition is required every 8 bits – to keep the data clock in sync with the signal. To fulfill the second requirement (and to play it safe), the transmitter inserts a dummy bit before every fourth transmitted bit (setting it to the reverse of the next bit). This dummy bit is stripped off and discarded by the receiver. The effective bit rate is thus 4*b*/5, where *b* is the nominal bit rate.

The checksum, independent of the encoding scheme, is the arithmetically negated sum of all 4-byte long-words in the packet. Thus, the sum of all long-words in a correctly received packet together with the checksum, should be zero.

The device defines the following control operations:

`RADIO_CNTRL_SETPRWAIT`

The argument points to a `word` whose contents are interpreted as an unsigned integer number indicating the amount of waiting time (in milliseconds) for a preamble. This will apply to all subsequent `READ` operation, which will spend up to that much time waiting for a high-signal segment of a preamble before giving up and assuming that there is no packet to receive. The default value is 4. This parameter is irrelevant for the XEMICS board where reception events are explicitly triggered by received preambles. For XEMICS, the `READ` operation fails, if no preamble is immediately perceptible when the operation is invoked.

`RADIO_CNTRL_SETPTRIES`

The argument points to a `word` indicating the number of tries to synchronize to a received preamble before giving up (and assuming that the `READ` operation has failed). Having recognized a 3-*bit* long high segment of a preamble, the receiver will expect a short low segment followed by another long segment, and so on, until it sees a 2-*bit* high segment ending the preamble. If the preamble structure does not match this pattern, the receiver will try to resynchronize on the next 3-*bit* long segment. The parameter specifies the limit on the number of such resynchronization attempts. The default value is 4.

Regardless of the setting of this option, to be recognized as valid, a preamble must include at least 2 full, undisturbed, long-short cycles.

This parameter is irrelevant (and ignored) for the XEMICS board, which synchronizes to the preamble internally.

`RADIO_CNTRL_SETPRLEN`

The parameter points to a `word` indicating the length of a transmitted preamble (it applies to the transmitter, i.e., all subsequently transmitted packets will have preambles of this length). For the XEMICS board, the preamble is an alternating pattern of ones and zeros ending with two consecutive ones. One unit of length corresponds to a full cycle, i.e., two consecutive bits (1/0). The minimum length (required to recognize the preamble at the receiver and tune the data clock to the received packet) is 16 cycles and the default length is 24.

For the other boards (using *biphase* encoding), one preamble cycle consists of a long high signal followed by a short low signal. The default preamble length is 32 cycles in that case.

`RADIO_CNTRL_XMTCTRL`

The parameter points to a single character. If the value of this character is nonzero, the transmitter is enabled. Otherwise, the transmitter is disabled.

Note that the transmitter must be enabled before `WRITE` as otherwise the operation will be void. The explicit enable operation separate from `WRITE` is to account for those boards that require a non-trivial warm-up procedure before commencing a physical transmission. Following a single enable operation, a number of frames can be sent in sequence before the transmitter is disabled.

`RADIO_CNTRL_RCVCTRL`

The parameter points to a single character. If the value of that character is nonzero, the receiver is enabled. Otherwise, the receiver is disabled.

On those boards on which the entire device can be disabled (e.g., to conserve power), disabling both the transmitter and the receiver disables the device. Enabling the transmitter while the receiver is enabled, temporarily disables the receiver, which will be re-enabled when the transmitter becomes disabled again. Enabling the receiver while the transmitter is enabled has no immediate effect, although the enabled status of the receiver is stored. The receiver will be physically enabled when the transmitter becomes disabled.

`RADIO_CNTRL_SETPOWER`

The parameter points to a word indicating the new level of transmitted power. This operation is void except for `RADIO_XEMICS`, where the transmitted power is determined as one of four discrete levels from 0 (the lowest power, but not actually zero) to 3 (the maximum power). The default power assumed for the XEMICS board is 3.

`RADIO_CNTRL_CHECKSUM`

If the first byte of the buffer parameter is zero, the operation disables checksum insertion by the transmitter and its verification by the receiver. Also, the restrictions on the packet length (see above) are completely relaxed. Any sequence of bytes can be transmitted by `WRITE`, and any sequence of received bytes will be returned by `READ` with no verification. If the first byte of the buffer is nonzero, the default checksum processing (with packet length restrictions) is resumed.

`RADIO_CNTRL_CALIBRATE`

The argument points to a word specifying the required raw bit rate in physical *bits* per second. The operation recalibrates the device to the specified rate. Different ranges of values may be acceptable by different radio boards. Generally, those values are between 4800 and 38400 inclusively, with the XEMICS board only accepting discrete values: 4800, 9600, 19200, and 38400.

Some values may not work with the other boards. The defaults are 16000 for RADIO_VALERT, 18000 for RADIO_RFMI, and 19200 for RADIO_XEMICS.

RADIO_CNTRL_READPOWER

There is no argument. The operation returns an integer value indicating the received signal power last perceived by the receiver. At present, this only works for the XEMICS board, with the received power quantized into four levels 0-3. For the remaining boards, the operation always returns 1.

RADIO_CNTRL_READSTAT

There is no argument. The operation returns an integer value indicating the number of milliseconds that have elapsed since the receiver has spotted last activity in the ether. Depending on the board, the mechanism for detecting this activity may differ (and the value returned by the operation may be more or less misleading). If the returned value is MAX_INT (0x7fff), it conceptually means *infinity*.

For the activity status (RADIO_CNTRL_READSTAT) to be available at all, the kernel must compiled with RADIO_INTERRUPTS != 0 (see sysio.h). This option indicates that the radio interface is either capable of triggering interrupts (like in the XEMICS case), or that it receives some assistance from the clock interrupt service routine (for the other two boards). In the latter case, the system monitors (at millisecond intervals) the received signal level and simulates interrupts when the signal is high. This monitoring is only carried out when the receiver is enabled.

When RADIO_INTERRUPTS != 0 (which is forced for XEMICS and optional for the other boards), the following operations become available:

```
---> bool rcvwait (word state)
```

With this operation, a process willing to receive a packet may declare that it wants to be awakened in the indicated state, when a reception event (simulated or physical) is triggered. At most one such event can be awaited at any time. The function returns NO (and is ineffective), if some process is already waiting for the event; otherwise, it returns YES.

For the boards with simulated reception events (not for the XEMICS board), the value of RADIO_INTERRUPTS determines whether a single sample of a received signal is sufficient to trigger the simulated reception event. This is the case when RADIO_INTERRUPTS == 1. Otherwise, at least two such samples are required, and then RADIO_INTERRUPTS gives the maximum number of milliseconds separating such samples.

```
---> void rcvcancel (void)
```

The operation cancels the effects of `rcvwait`. This also happens automatically when the process is awakened, but only when it is awakened by the reception event. Otherwise, the event remains pending until triggered (and may wake up the process later).

The arguably clumsy implementation of the above tools has been dictated by the reasons of efficiency (and the fact that in some cases they directly interact with the critical clock interrupt routine). It is not recommended to use them directly from the application. It is assumed that the raw radio interface will be covered by a TCV plugin which will turn it into an asynchronous, buffered, packet-oriented networking interface.

8.6 ADC

For efficiency, the Analog to Digital Converter is not accessed via the standard io operation, but with the following special functions:

```
---> void adc_start (int mode, int mask, int interval)
---> int adc_read (int which)
---> void adc_stop (void)
```

The first function initializes the acquisition of data. Its first argument specifies one of the following modes:

`ADC_MODE_TEMP`	collect the temperature from the on-chip sensor
`ADC_MODE_VOLTAGE`	collect the power supply voltage
`ADC_MODE_INTREF`	up to four inputs compared to internal reference voltage
`ADC_MODE_OUTREF`	up to three inputs with the internal reference sent out over A3
`ADC_MODE_EXTREF`	up to three inputs with external reference arriving on A3
`ADC_MODE_DIFFER`	up to two differential inputs (A0-A1 and A2-A3)

The `mask` argument of `adc_start` specifies which of the possibly multiple values should be actually collected. It is ignored for the first two modes, as each of them indicates a single value. For example, if the `mask` is `0x3` (bits 0 and 1 set) and the mode is `ADC_MODE_OUTREF`, the first two of the three values (i.e., A0 and A1) will be collected. The `mask` of `0xf` always selects everything.

The third argument specifies the sampling interval in milliseconds. Every sampling interval, the collected samples are extracted from the ADC and stored internally by PicOS to be available for retrieval by `adc_read`. The new samples overwrite the previously stored values.

The argument of `adc_read` indicates one of the samples selected by the `mask` of `adc_start`. Its value must be less than the number of bits in the `mask`. The

returned sample value is a signed integer number between -2048 and +2047 inclusively. `adc_stop` terminates the collection.

8. Library functions

In addition to the functions built into the kernel, some useful operations are available in the library (directory `Platform/Lib`). They can be referenced after including the respective header files (`.h`). Note that the library directory is automatically searched for application includes.

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

Requires `form.h`. This function uses the specified format (`fmt`) to interpret its remaining arguments and encode them into the buffer `buf`. If `buf` is `NULL`, the buffer is 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 `CODE_LONG_INTS` option (see `options.sys`) is set, '`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 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`' prefix is applicable to the numerical descriptors. 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).

---> int ser_out (word state, const char *msg)

Requires ser.h. The function writes the specified message to the currently selected UART (as illustrated in the example in section 4). If state is not NONE, the function returns when the request has been accepted (and then it returns zero). Otherwise, the process is put to sleep and resumed in the indicated state when it makes sense to retry the operation. If state is NONE, the function returns immediately, even if the request has not been accepted. In such a case, the function returns a nonzero value that can be interpreted as the indication of reason. Any value other than NONE is the Id of a process that must finish before the operation will be acceptable (it is the previous incarnation of the writing process started by a previous call to ser_out or ser_outf). NONE means that the function has failed on the lack of memory to accommodate the output buffer.

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

Requires ser.h. As ser_out, but the 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 127 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. A line is terminated by LF or CR, whichever character comes first, and the terminating character is not stored. An LF character occurring at the beginning of a line is ignored. 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, the function returns the Id of the process (the previous reader in progress) 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` and `scan`. If `state` is not `NONE`, the function returns the number of items that have been extracted from the input line. Otherwise, its value is interpreted as for `ser_in`.

```
---> int dsp_lcd (const char *msg, bool kill)
```

Requires `lcd.h`. This function displays the specified message on the LCD. It returns immediately delegating the actual operation to a special process. If the message is longer than 32 characters, it will be periodically scrolled through the LCD at the rate of about 1/2 second per character.

If a message is currently being displayed, i.e., the display process is active, and `kill` is `NO`, the function returns -1 and does nothing. If `kill` is `YES`, the previous process is killed and a new one is started. If the message doesn't have to be scrolled (i.e., it is 32 characters or less), the display process terminates as soon as the message has been written to the device. Otherwise, it stays alive until killed by another call to `dsp_lcd`. The right way to clear the display (and get rid of the display process) is to execute `dsp_lcd ("", YES)`. Upon success, `dsp_lcd` returns zero.

```
---> void upd_lcd (const char *msg)
```

This is an alternative to `dsp_lcd` recommended when the output string is no longer than 32 characters (i.e., within the LCD capacity) and when only some fragments of the string tend to change. The function writes the string using the minimum number of changes to its already displayed previous version (using `upd_lcd`, not `dsp_lcd`), which makes the update appear smoother.

```
---> void dsp_led (const char *pat, word intvl)
```

Requires `led.h`. The function interprets the specified character string as a series of patterns to be displayed in the four LEDs present on the board. Each character is interpreted as a hexadecimal digit representing a 4-bit value, with zero substituted for any character that is not a valid hexadecimal digit. The pattern is advanced every `intvl` milliseconds wrapping around at the end.

The function returns immediately having spawned a process to take care of the pattern switching. If such a process already exists, it is killed and a new one is started. If the specified message is `NULL` or an empty string, any present display process is killed and no new process is started.

```
---> void beep (word del, word tone, word state)
```

Requires `beeper.h`. The function sounds the beeper for `del` seconds using the indicated tone. While the beeper is buzzing, the caller process remains blocked and then it transits to the specified state. The value of `tone` should be between 1 and 6 inclusively.

9. Process tracing

By including the file `trc.h` from `Platform/Lib` in the header of an application file, one redefines the standard operations (macros) `process` 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.

10. Memory allocation

PicOS is equipped with a rather naive 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 present in the system (see `options.sys`), the heap is located entirely in SDRAM. Otherwise, it uses whatever on-chip RAM remains available after accounting for all static variables. Note that constants (declared with the `const` keyword of C) are allocated in code and take no RAM space.

The application 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 application program. 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 application. If TCV is configured into the system, pool 1 covers the packet storage of TCV (and is not used for anything else). More pools can be defined by the application, but the application should never use pool 1 if TCV is present.

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

```
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. The following operation:

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

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

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

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

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, and the second 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 (in `options.sys`), 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) 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 in fact 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`, it carries out 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. Spare SDRAM

If the SDRAM option is compiled in, PicOS maps the first 32K words of SDRAM into the data space starting at address `0x4000`. This area is then used for dynamic memory allocation (i.e., `malloc/free`). The total amount of SDRAM on the evaluation board is 16MB (8M words), which is considerably more than what can be referenced by the 16-bit data address. Paging on the eCOG is not very easy to implement because there are only two rather inflexible translators, and, as SDRAM must be mapped in the middle of the data address space, both translators are needed to map the single (preallocated) 32K block. Nonetheless, PicOS provides a way to use the remaining (spare) SDRAM of 8M - 32K words as a relatively straightforward storage area that can be accessed with the following two operations (provided directly by the kernel):

```
---> void ramput (lword dest, address src, int len)
```

This operation moves `len` words from the location pointed to by `src` to spare SDRAM at position indicated by the first argument. For the purpose of this operation, the spare SDRAM (i.e., total SDRAM minus the first pre-allocated 32K block) can be viewed as an array of words indexed by numbers from 0 (inclusively) to 8M - 32K (exclusively). The limit is available as `SDRAM_SPARE`. The first argument of `ramput` can be thus viewed as an index into the spare SDRAM area.

```
---> void ramget (address dest, lword src, int len)
```

This is the matching ‘get’ operation for fetching chunks of data from spare SDRAM.