PicOS

Version 2.03
November, 2008

Preamble

PicOS is a small-footprint operating system for organizing multiple activities 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.

1. 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 relevant 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 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

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

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 is accomplished with VUE[2] (Virtual Underlay Emulation Engine), which is a SIDE-based emulator for wireless praxes programmed in PicOS.

2. 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, typically 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.

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' 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

[3]This is PicOS's term for “application.” Owing to our idiosyncratic paradigm of developing and integrating such applications, we prefer to use a name that tells this endeavor apart from “traditional” application development.

process: in such a case, each instance will operate on its own local data, in addition to being able to access all global data (which is shared). 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 local (implicitly declared) variable **data** (note that its 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 sniffer strands. 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 can usually exist 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 essentially aliases for simple variants of **process** and **endprocess**.[4] However, consistent adherence to the two aliases, and avoidance of **process** and **endprocess**, helps to make a PicOS praxis compatible with VUE[2].

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 function is executed at state 0. Thus, state 0, which must always be present, is the initial state. In particular, symbol **RC_TRY** in Figure 1 is assumed to be defined as 0.

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

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 forgotten. 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]

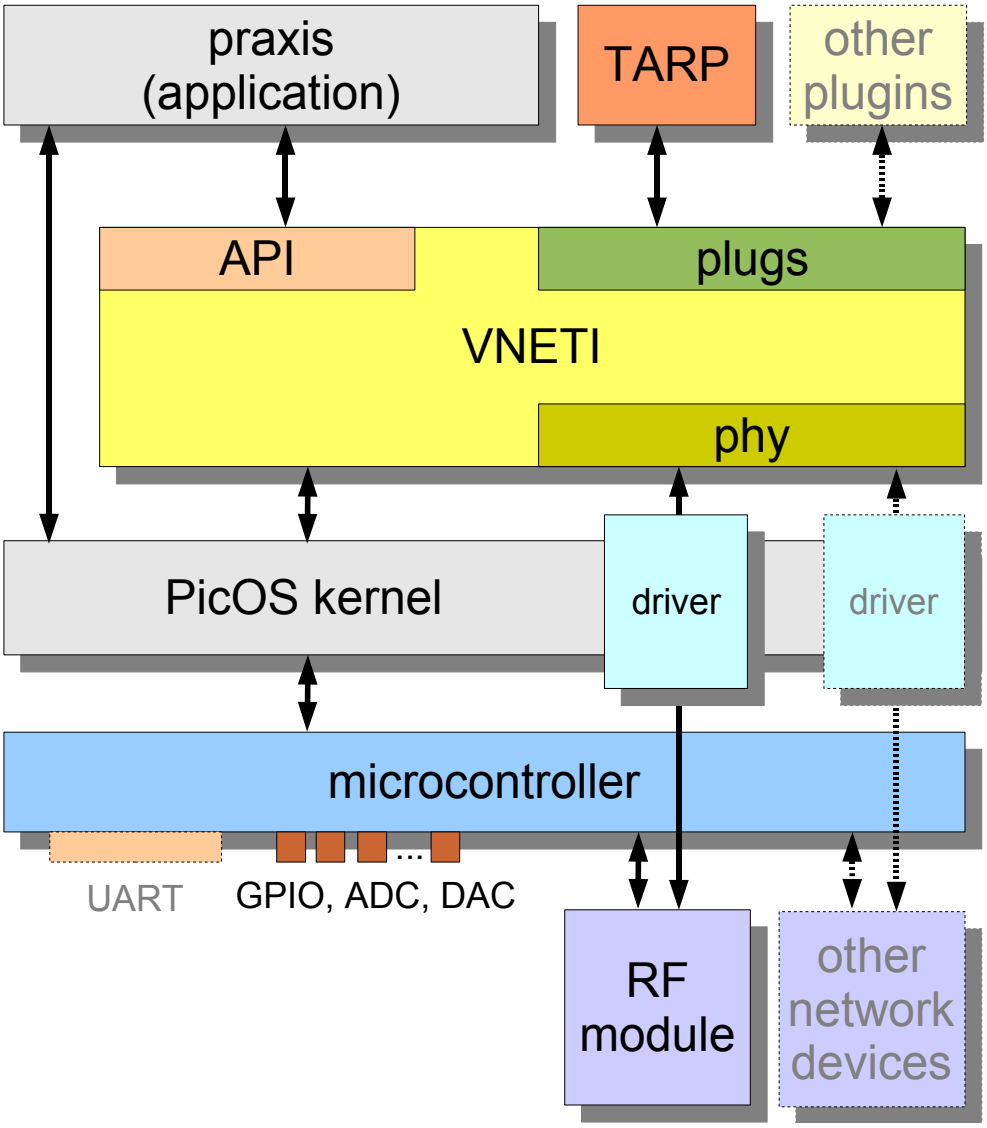

Figure 3. System structure.

3. 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

[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.

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.

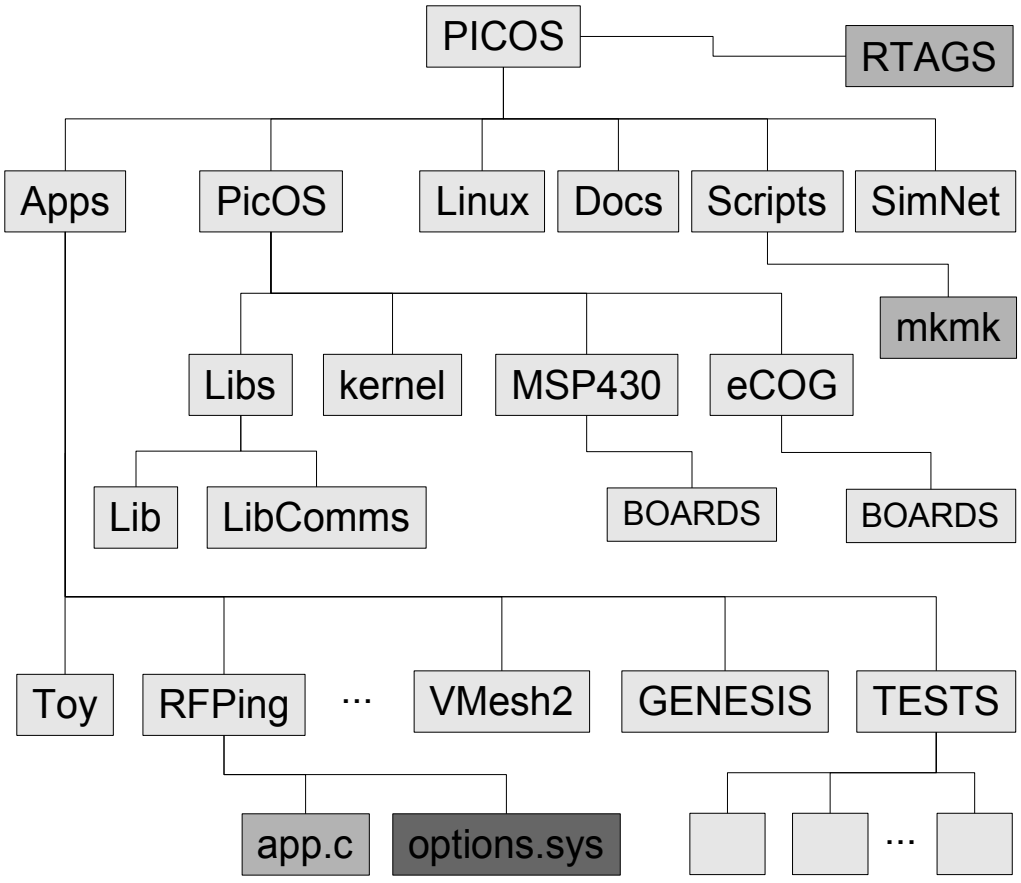

Figure 4. PicOS directory tree.

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 two 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).

  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 the argument to *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*, which unpacks into MspFet and Terminal – two handy programs mentioned in the next section.

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 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 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.

4. Software co-requisites

The following software (available for free) is required to compile, load, and debug PicOS praxes under Windows XP. MSP430 development is also possible 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 MspFet and Terminal. Both programs can be immediately run under Windows (no installation is required). MspFet is a self-explanatory JTag programmer for MSP430, which may be useful for loading programs into the microcontroller. 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).

5. Quick start

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 v2.01, Copyright (C) Olsonet Communications, 2002-2006
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.

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 v2.01, Copyright (C) Olsonet Communications, 2002-2006
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.

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:

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

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

```
#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 2, the argument 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*.

6. 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*.

7. 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.

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.

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

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.

void snooze (word state)

This operation makes it possible to continue an interrupted **delay** wait. Suppose that a process issues a wakeup 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). Essentially, **snooze** has the same semantics as **delay** with the first argument equal to the leftover time from the last call to **delay**. This semantics 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 snooze (word state)

This operation makes it possible to continue an interrupted **delay** wait. Suppose that a process issues a wakeup 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). Essentially, **snooze** has the same semantics as **delay** with the first argument equal to the leftover time from the last call to **delay**. This semantics 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.

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

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

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.

long seconds (void)

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

void ldelay (word nmin, word state)

One problem with **delay** is that the maximum waiting time is 64 seconds. This function can be used to wait for a long amount of time specified in *minutes*, where one *minute* equals 64 seconds (64 x 1024 PicOS milliseconds). This waiting time is inaccurate in that a process waiting on ldelay is always awakened at a *minute* boundary according to the indications of the seconds clock. Suppose that the seconds clock shows 9016, and a process executes **ldelay(2,WakeMe)** followed by **release**. The waking event will occur at second 9088, which is the nearest minute boundary with the property that the next minute boundary would take us beyond the requested delay. Thus, expressed in seconds, the actual delay can be anywhere between (**nmin** – 1) x 64 + 1 and **nmin** x 64, inclusively.

word ldleft (int pid, word *sec)

The function returns the number of 64-second *minutes* that the process identified by **pid** will wait before its **ldelay** timer elapses. If **sec** is not **NULL**, it points to the location where the residual number of seconds will be stored. If the process is not waiting on **ldelay**, the function returns **MAX_UINT**, i.e., 65,537.

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 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.

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 pointed to by 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. For the 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. The maximum size of the timer pool is 4 entries.

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.

udelay (word del)

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. 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).

[9] This applies exclusively to eCOG1.

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).

8. 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.

[10] At present, the UART rate on eCOG cannot be changed (it is "hardwired" at 9600 bps). Thus, this fragment applies exclusively to MSP430. 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**.

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
```

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).

9. 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:

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

```
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)
```

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.

10. Power conservation tools

A few functions provided by the kernel make it possible for the praxis to save battery power during periods of extended idleness, without losing track of time and/or the interesting external events. Here is the list of those operations:

```
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 between 16 to 1 tick per second, depending on the crystal rate

and, possibly, some other system options. Time measurement (in seconds) is still accurate, and **delay** operations work, except that they are rounded to the much coarser grain.

```
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** results in slowing down the clock (implies **clockdown**) and selects a deeper sleep state during the periods of CPU inactivity. In particular, on MSP430, **powerdown** selects the LPM3 sleep mode, whereby 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 increased with respect to the normal power-up mode (in which the sleep state is the least deep of them all).

```
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.

11. 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 lhold (word state, lword *sec)
```

Requires *lhold.h*. The function implements precise (long) delays for the prescribed number of seconds (see the system functions **delay** and **ldelay**). The argument points to a **lword** storing the number of seconds to wait. This number is changed by the function and thus it is passed through a pointer. Here is how the function should be called:

```
...
static lword del;
...
    del = 3600;
  entry (HOLD_ME)
    lhold (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 and monitor the progression of waiting time.

It may be worthwhile to 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.

12. 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.