olsonet Communications

Programming under

P i c O S

Version 5.4
August 2017

Table of contents

Related documents:

[installation]	Installation and Quickstart
[vuee]	VUE[2] the Virtual Underlay Execution Engine
[side]	SIDE/SMURPH: a Modeling Environment for Reactive Telecommunication Systems
[picomp]	PiComp: the PicOS Compiler
[vneti]	VNETI: Versatile NETwork Interface
[pip]	PIP: an Integrated SDK for PicOS
[mkmk]	mkmk: the PicOS Makefile Maker
[serial]	UART Communication via VNETI (TCV)
[lcdg]	LCD-G interface
[oss]	Generic, simple OSS interface
[oep]	Object Exchange Protocol
[eol]	EEPROM object loader

1 Introduction

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, orthogonal, and expressive tools for event-driven I/O and inter-process communication.

So far, PicOS has been implemented on three CPU architectures: eCOG1 by Cyan Technologies[1], MSP430 CPU series by Texas Instruments[2], and ARM Cortex (represented by CC1350 by Texas Instruments).

In comparison with TinyOS[3], PicOS has the following advantages:

1. All of the program dynamics available to the programmer is captured by PicOS's threads (finite state machines) rather than interrupt service routines (or callbacks). As all threads share the same (global) stack, PicOS's stack has no tendency to run away.

2. PicOS is incomparably more flexible with memory. Dynamic memory allocation (even within devices with less than 1K or RAM) is its essential feature.

An important component of the complete PicOS platform is VUEE (also known as VUE[2]) – the Virtual Underlay Execution Engine, which is also described separately (see [vuee]

[1] This architecture is now extinct and support for it has been retired.
[2] See http://www.ti.com/.
[3] See http://docs.tinyos.net/.

and [picomp]). VUE^2 makes it possible to execute PicOS praxes in a virtual, multi-node, wireless environment.

2 Finite State Machines

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 every task (process, thread). The most critical example of such a resource is the stack space: 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 (and inconveniently) limited, it still remains a significant component of the total amount of memory resources needed to describe and sustain a task in the system. With the 2KB of RAM on MSP430F148, 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. By enforcing the FSM appearance of a task, PicOS stimulates its clarity and self-documentation, which is especially useful and natural in 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 [side]), which is a specification and simulation environment for reactive systems (mostly telecommunication systems). The fact that PicOS is closely related to SIDE makes it possible to execute PicOS programs in a realistic virtual environment created in SIDE. This brings us VUE^2 (Virtual Underlay Emulation Engine), which is a SIDE-based emulator for wireless praxes programmed in PicOS [vuee].

2.1 Hello World

Here is a complete “hello word” program. It writes a short message to the UART and then quits:

```
#include "sysio.h"
#include "ser.h"

fsm root {
      state START:
            ser_out (START, "Hello World!!\r\n");
            finish;
}
```

The program consists of a single function-like construct which we call a *finite state machine*, or an FSM for short. FSMs are sometimes called *processes* (also *threads* or *strands*) because, like processes, they can be started (run) and terminated (killed). A single FSM (viewed as a piece of code) can have a number (possibly more than one) of dynamic instances (processes) running at the same time. The root FSM is automatically

run by the system (in one instance) immediately after reset. It plays a role analogous to function *main* in a straightforward C program.

The language of PicOS is C with a few added keywords and constructs. A program in PicOS is preprocessed into C by a Tcl script named PiComp. The same script (invoked with different arguments) precompiles PicOS praxes into VUE^2 models.

An FSM definition begins with the keyword **fsm** followed by a name, which must be unique within the entire program (note that a program may comprise several C files). Sometimes the FSM name is followed by an argument, like in this example:

```
fsm output (const char *msg) {
        state WRITE_MSG:
              ser_out (WRITE_MSG, "I am ready!!\r\n");
        state WAIT_FOR_SIGNAL:
              when (getcpid (), WAKE_UP);
              release;
        state WAKE_UP:
              ser_outf (WAKE_UP, "Time %lu, %s\r\n",
                    seconds (), msg);
              proceed WAIT_FOR_SIGNAL;
}
```

Such an argument, if present, must declare a single variable of a simple type (e.g., an integer or a pointer) which refers to some object that the FSM is expected to handle. This is the so-called data attribute of the FSM. Using the process analogy, we would say that that object represents the private data of the FSM, so we can have multiple FSMs running the same code while operating on different data. Here is a sample root FSM that references the one above:

```
sint out1, out2;
...
fsm root {
        state START:
              out1 = runfsm output ("message 1");
              out2 = runfsm output ("message 2");
        state DELAY:
              delay (1024, SIGNAL1);
              release;
        state SIGNAL1:
              trigger (out1);
              delay (1024, SIGNAL2);
              release;
        state SIGNAL2:
              trigger (out2);
              proceed DELAY;
}
```

The two FSMs together (with the addition of some standard headers) constitute a complete program. The root FSM starts by creating two instances of the output FSM, passing each of them a slightly different string to handle. Each of the two copies will say "**I am ready!!**" upon activation and then wait for a signal (equal to its respective numerical process ID). Having received such a signal, the FSM will output its assigned string preceded by the number of seconds elapsed since the startup (i.e., system reset), and wait for another signal. The root FSM will be delivering those signals at one second

intervals, every other signal being addressed to the other instance of the **output** FSM. In consequence, you will see this sequence of lines on the UART:

```
I am ready!!
I am ready!!
Time 1: message 1
Time 2: message 2
Time 3: message 1
Time 4: message 2
…
```

appearing at 1 second intervals.

Before we go into details, let us try to get some quick intuition backed by the programming constructs illustrated in the above examples. The code of an FSM is organized into a number of *states* representing points where the FSM can be entered (activated). Those states receive symbolic names that can be referred to by some operations. For example, the **when** statement (in the **output** FSM) accepts two arguments: the first one describes an event, the second indicates the state to be assumed by the FSM when the event occurs.

Once an FSM has been activated (its code entered at a particular state) it will remain active (holding on to the CPU) until it gets into a situation when it has to wait for something to happen. For one illustration, consider the call to **ser_out** (in the **output** FSM). This is a library function implementing the operation of writing a piece of text to the UART. Such an operation may block (e.g., if it has to wait for the completion of a previous activity on the UART), or it may succeed (and simply pass through), if the output data can be accepted right away. The latter case is uninteresting: the function will just return. Should the operation block, the FSM will be forced to exit, i.e., release the CPU with the intention of regaining it later when it makes sense to try the blocked operation again. When that happens, the FSM will be resumed in the state specified as the first argument of **ser_out**. This will have the effect of re-executing (re-attempting) the previously blocked operation. In the meantime, i.e., while the output FSM is waiting for an opportunity to send its data to the UART, the system is free to assign the CPU to another FSM.

One important thing to remember is that all situations when an FSM may block (waiting for some event or condition) *must* involve a state. This is because an FSM can only resume its execution at a state boundary (regardless of the reason why it lost the CPU in its last turn). Some operations (library functions or system calls) implement the blocking internally. An FSM can also block explicitly by issuing one or more *wait requests* and then *releasing* the CPU directly. This is, for example, what the **output** FSM does after executing **when** (in state **WAIT_FOR_SIGNAL**), which illustrates a wait request. Note that **when** itself doesn't block: it only declares a waking condition (event). In particular, you can issue several wait requests (declaring alternative waking events) before releasing the CPU (via the operation **release**). Another example of a wait request is **delay** (in the root FSM), which sets up a timer to wake up the FSM after the prescribed number of clock ticks. Also note that, unless it does something explicit, an FSM will freely fall through the state boundaries. For example, if **ser_out** in state **WRITE_MSG** (in the **output** FSM) doesn't block (returns), the FSM will immediately find itself executing **when** in the next state (**WAIT_FOR_SIGNAL**). Operation **proceed** provides for a direct switchover to the indicated state (not necessarily the one immediately following the current state) without having to wait for any events.

2.2 Basic types

The essential collection of PicOS-specific types, functions, macros and constants is defined in file *sysio.h,* which must be included by any PicOS praxis. In addition to the standard types provided by C, PicOS defines the following basic types:

word	a 16-bit unsigned integer
sint	the preferred standard integer type for the CPU (basically equivalent to **int**)
lword	32-bit unsigned integer
lint	32-bit signed integer
address	representing a generic pointer, which in PicOS is a pointer to a **word**; thus, **address** is equivalent to **word*** [4]
aword	the unsigned integer type whose size coincides with that of **address**; on 16-bit architectures, **aword** is equivalent to **word**, on 32-bit architectures, it is equivalent to **lword**
byte	equivalent to **unsigned char** and intended for representing raw bytes
bool	if this type is not defined by the compiler, PicOS makes it equivalent to **byte**
Boolean	equivalent to **byte**
fsmcode	this is a function pointer type referring to FSM functions

One of the reasons for introducing aliases for some of the simple standard types is the need to maintain a general portability of the praxis source code, and most importantly its VUE^2-compliance, i.e., direct acceptance by the VUE^2 compiler and the emulator. This is not required for a target (microcontroller) platform; however, to make sure that your praxis is VUE^2-compliant, you should never use types like **int** or **unsigned int** directly.

The following symbolic constants (defined by PicOS) describe the ranges of values represented by some of the above types:

#define MAX_WORD	**((word)0xffff)**
#define MAX_LWORD	**((lword)0xffffffff)**
#define MAX_AWORD	the maximum value represented by **aword**
#define MAX_SINT	the maximum (positive) value of machine's int
#define MIN_SINT	the maximum (negative) value of machine's int

Below we list a few symbolic names assigned to some special constants (see *sysio.h*):

```
#define     NULL        0
#define     NONE        ((word)(0xffff))
#define     LNONE       ((lword)(0xffffffff))
#define     LWNONE      LNONE
```

[4] This “feature” is a legacy of the first implementation of PicOS for the eCOG1 of Cyan Technologies. Pointers to bytes (or **void** pointers) on the eCOG1 were special, clumsy, cumbersome, and inefficient.

```
#define       WNONE         NONE
#define       BNONE         0xff
#define       ERROR         NONE
#define       BLOCKED       (-2)
#define       NO            ((Boolean)0)
#define       YES           ((Boolean)1)
```

The most useful of them are **NULL**, **WNONE**, **NO**, and **YES**.

2.3 FSM syntax

One confusing aspect of the informal terminology introduced in section 2.1 was the lack of distinction between the description (code) of an FSM and its dynamic (running) incarnation. From now on, when we say “FSM” we shall mean the static (textual) definition of an FSM (i.e., a piece of code beginning with the **fsm** keyword). Whenever it is relevant to the discussion, an FSM that specifies a data argument (like the **output** FSM from page 5) will be referred to as a “strand FSM”, in contrast to a “thread FSM”, i.e., one that declares no private data. A dynamic instance of an FSM will be called a *process* (the general case), or a *thread* or *strand* (if we want to differentiate between the two FSM types).

Processes are identifiable by numbers of type **aword** called *process identifiers* (or PIDs), assigned (by the system) at the moment of their creation (this is the value returned by **runfsm**) – see page 5). A process is created explicitly (by another process) and can be terminated either by itself or by another process. One process, described by the root FSM, is created and started automatically by the system after reset. This means that every praxis must define a root FSM, which is responsible for creating all other user processes. Note that the root process need not exist forever, e.g., it can start up the (other) processes needed by the praxis and terminate itself. Some processes used by the system (e.g., belonging to device drivers), may be started internally before the root process.

An FSM header has one of the following forms:

```
fsm fsm_name {
fsm fsm_name (simple_type_name variable) {
fsm fsm_name (simple_type_name) {
fsm fsm_name () {
```

with the last variant being strictly equivalent to the first. The first and last variants define a thread FSM, while the middle two variants define a strand FSM (i.e., one that operates on a private data object).

The name of an FSM (***fsm_name***) can be any legitimate (and unique) C identifier. FSM names are always global within the scope of the entire program (across all C files), i.e., you cannot have a **static** FSM.

With the second variant, the specified variable becomes visible to the FSM body in the same way that a function argument is visible to the function body. Its declared type must be simple enough to fit **aword** (see page 7). Thus, it can be, e.g., an integer type, a character, or a pointer, but it cannot be a structure. The easy formal rule is that the variable must allow itself to be cast to **aword** in a C statement. Array types are illegal (because they cannot be used in casts) , e.g.,

```
fsm foobar (word A []) {
```


will not work, although the pretty much equivalent:

```
fsm foobar (word *A) {
```

will.

If the variable name is missing (variant 3), it defaults to **data**.

The name of an FSM can be referenced by some operations. One such operation is **runfsm** (see page 5) which creates and runs a process. For that, the name must be visible in the current scope, pretty much in the same way that a C function must be visible at a point of reference. If the actual FSM is defined elsewhere, e.g., in a different file, or further down in the present file, it can be declared (announced) before usage in a way similar to announcing regular functions, e.g.,

```
fsm output (const char*);
```

You can list multiple headers per announcement separated with commas, e.g.,

```
fsm output (const char*), monitor;
```

announces two FSMs, the second one being a thread. An empty pair of parentheses is the same as their absence, e.g., the above announcement is equivalent to:

```
fsm output (const char*), monitor ();
```

The general layout of an FSM is:

```
header {
      optional local declarations
      list of states
}
```

where *header* has one of the four forms listed above. This resembles the layout of a function, except for the requirement that the body (executable statements) be organized into a sequence of states, i.e.,

```
state state_name1:
      list of statements
state state_name2:
      list of statements
...
```

where the state names are C identifiers unique within the function scope of the FSM. The organization of the list of states is subject to the same rules as for the list of cases of a switch statement (except that there is no default). As a legacy feature, the keyword **state** can be replaced with **entry** (with exactly the same meaning).

The first state on the list is also the one to be entered automatically when the FSM's process is created and run for the first time (by **runfsm**). It is possible to explicitly declare any state as the initial one by putting the keyword **initial** in front of it, like in this FSM:

```
fsm toggle {
      state TOGGLE_LEDS:
```


```
            leds (0, 1); leds (1, 0);
            delay (2048, INVERT);
            release;
      state INVERT:
            leds (0, 0); leds (1, 1);
      initial state WAIT_FOR_TIMER:
            delay (2048, TOGGLE_LEDS);
}
```

When started, the FSM's process will find itself in state **WAIT_FOR_TIMER**. Note that there is no need to execute **release** at the end of that state, as there is no more code underneath. Also note that the **delay** statement in state **WAIT_FOR_TIMER** will be executed as a continuation of the code at state **INVERT** (state boundaries are fall through).

The maximum number of states that a single FSM may define is 4096. The state symbols are equivalent to (short) unsigned integer constants: they can be saved, e.g., in **word**-type variables, and those variables can be used instead of the symbols as arguments of all functions that expect state identifiers. This doesn't apply to **proceed** or **sameas** (page 15) whose arguments must be state symbols (as appearing at **state** statements). These operations are not functions, however. The initial state is always number zero. It is an error to explicitly declare more than one state as **initial**.

2.4 Local variables

An FSM is is treated by the system as a function that gets called on every activation of the FSM's process, with some hidden argument identifying the state to be assumed for that activation. Similarly, when the FSM's process gives up the CPU, the function simply returns (to the system). This means that there is no stack context expected to survive across the different activations of the same process. This is good news as it allows PicOS to run all its processes on the same single and global stack. However, this also means that any variables whose contents are supposed to survive state transitions cannot be allocated on the stack.

For this reason, any variable declared after an FSM header and before the first state is treated as if its declaration were preceded with **static**. This means that the variable is allocated in global RAM, but is only visible to this particular FSM (all its processes). Beware that such variables are *not* “local” to the FSM's process, because they are in fact *shared* by all the processes of the FSM. However, they do survive state transitions.

For illustration, consider this variant of the **toggle** FSM from section 2.3 in which we reduce the number of states by storing some state information in a variable:

```
fsm toggle {
      byte on = 1;
      state TOGGLE_LEDS:
            leds (0, on); leds (1, 1 - on);
            on = 1 - on;
      initial state WAIT_FOR_TIMER:
            delay (2048, TOGGLE_LEDS);
}
```

Note that if the **on** variable were treated as automatic (and allocated on the stack, as for a regular C function) the FSM wouldn't work. First of all, the variable would not retain its previous setting at state transitions (e.g., at subsequent activations of the process at state **TOGGLE_LEDS**), because its stack location would be overwritten by other

processes running in between; second, the variable would be reset to 1 at every activation (because of the initializer).

Also note that the interpretation of the **on** variable is basically the same as if the variable were global, while being visible only to the processes of this particular FSM. But still, a single copy of such a variable is effectively shared by all processes of the FSM, which is not always desirable. Quite often, as in the above case, it only makes sense as long as there's no more than one instance (process) of the FSM at any given time. For **toggle**, this assumption is probably natural, considering that the FSM operates on specific LEDs. If the variable is to be truly local, i.e., associated with exactly one process, and non-volatile at the same time, it should be put into the process's private data structure (and the FSM should become a strand), e.g.,

```
typedef struct {
      byte on;
} toggle_data_t;
...
fsm toggle (toggle_data_t *td) {
      state TOGGLE_LEDS:
            leds (0, td->on); leds (1, 1 - td->on);
            td->on = 1 - td->on;
      initial state WAIT_FOR_TIMER:
            delay (2048, TOGGLE_LEDS);
}
```

In a simple case, like the one at hand, this may look overly complicated. Note that the structure must be allocated and initialized before the strand is created, e.g.,

```
fsm root {
      ...
      state RUN_TOGGLE:
            toggle_data_t *p;
            p = (toggle_data_t*)
                        umalloc (sizeof (toggle_data_t));
            p->on = 0;
            runfsm toggle (p);
            ...
}
```

This may seem like a lot of effort. In a simple case, as the one above, you can avoid the structure (with its alignment overheads), by using a byte pointer instead, eg.,

```
fsm toggle (byte *on) {
      state TOGGLE_LEDS:
            leds (0, *on); leds (1, 1 - *on);
            *on = 1 - *on;
      initial state WAIT_FOR_TIMER:
            delay (2048, TOGGLE_LEDS);
}
```

and preallocating an array of bytes for all the multiple processes of **toggle**. Note, however, that a realistic FSM requiring a truly private data item is likely to need more than just a single byte. Also, there exist other simple and effective hacks that can be applied in trivial cases like the one above (see page 32).

Automatic local variables can still be declared within an FSM, but only in state blocks, as illustrated in the root process above (variable **p**). Such variables, by the very nature of their declarations, are only valid within the confines of the single state and must be re-initialized upon each activation at that state.

The data variable of a strand FSM has the status of a local variable that at every activation of the strand is set to the same value that was specified at the strand's creation (as an argument of **runfsm**, e.g., see page 5). The FSM can reset this variable, but the new value will only hold for as long as the strand retains control of the CPU. There is a way to make the change permanent in those rare cases when this is truly needed or useful (see page 32).

2.5 Process execution and CPU scheduling

When activated, a process typically issues a number of *wait requests* identifying the conditions (e*vents*) to resume it in the future and associating states with those events. Several events may be awaited by a single process at the same time. For example, consider this FSM fragment:

```
...
      state WAIT_FOR_DATA:
            when (DATA_READY_EVENT, GET_IT);
            when (QUIT_EVENT, STOP_IT);
            delay (4096, HEART_BEAT);
            release;
...
```

In this state, the FSM issues three wait requests before releasing the CPU: two for some “named” events and one for the timer. Having executed **release**, the process will become dormant until:

1. the **DATA_READY_EVENT** event occurs (is triggered), in which case the process will be activated in state **GET_IT**, or

2. the **QUIT_EVENT** event is triggered, in which case the process will be activated in state **STOP_IT**, or

3. the timer goes off (4 seconds later)

whichever of the three conditions materializes first. Thus, the multiple wait requests aggregate as an *alternative* of waking conditions. Note that the process can issue them in any order (which is immaterial) possibly mixing the statements with other operations. All those wait requests only become relevant when the process gives up the CPU (executes **release**). Then they will describe the process's continuation, i.e., the dynamic state transition function of the FSM (driven by events).

When a process is activated, its list of wait requests is initialized to “empty” (regardless of what the process was waiting for before getting into its present state). Thus, upon every activation, the process specifies its waking conditions from scratch.

Some system functions hide the operation of waiting and blocking the invoking process by performing **release** internally; however, they cannot completely hide the operation of unblocking (activating) the process when the awaited condition is met. This is because activation always involves a state; thus, a function that may potentially block must offer a state argument to the caller (e.g., see the call to **ser_out** on page 5).

Activities of multiple processes are coordinated by the CPU scheduler. Formally, the scheduler is non-preemptive, which means that a process receiving the CPU can only loose it by its own action and of its own will. That happens when the process executes **`release`**. Note that **`release`** can be executed by a function, e.g., a library function (so it need not be always plainly visible).

The scheduler is extremely simple, but it works fine for the kind of reactive processes you will find in a PicOS praxis. Such processes are dormant most of the time and only become active (usually for a very short time) when activated by events.

The dynamic pool of processes known to the system is described by a collection of fixed-size data structures known as PCBs (for Process Control Blocks). The PCBs of all processes currently present in the system are linked together into a single list.

One important attribute of a PCB is the *status word*, which indicates whether the process is *waiting* for an event or *ready* (i.e., willing to use the CPU). In the latter case, the status word also identifies the state in which the process should be run. When a waiting process receives one of its awaited events, its status changes to *ready*. Note that this does not automatically cause the process to receive the CPU. First of all, multiple processes may become ready at the same time (e.g., they may all have been waiting for the same event), while the CPU can only be assigned to one process at a time. Second, the scheduler is non-preemptive, which means that a process becoming ready while some other process is running (which may happen by some action of the running process, e.g., via **`trigger`**) will have to wait until the running process is done with the CPU.

Whenever the CPU becomes available and the system suspects that a process may be ready, it will execute a turn of the scheduler loop. The loop scans through the process list looking for the first PCB whose status is *ready*. If such a PCB is found, the process described by it is given the CPU. This consists in invoking the process's FSM function at the state prescribed by its activation event. If no ready process is found, the scheduler parks the CPU on a HALT instruction awaiting a system event (an interrupt). A system event causing a process to change its status from *waiting* to *ready* un-parks the CPU, which simply means that the CPU scheduler will execute another turn of its loop.

Note that hardware interrupts (which are normally invisible to the application programmer) can occur asynchronously with process activities; however, the process activities are in fact synchronous. If two or more processes are activated (made ready) simultaneously (e.g., by the same event), the order in which they will be run is determined by the ordering of their PCBs in the process list, which is always scanned in the same way. The ordering of processes (their PCBs) in the list tends to correlate with their creation order.

There exist two options (selectable at system compilation time) regarding the ordering of processes in the system's process list. With the first (default) option (see page 27), a newly created process is appended at the end of the list; thus, an older process has a priority over a newer one. The other option reverses that order by putting new processes at the head of the list.[5] If the praxis is truly reactive (and processes never engage in extensive calculations) their priorities are unimportant and can be assumed to be random. Thus, it is fair to say that when two processes are activated at the same time, the order of their actual execution is non-deterministic (especially if we don't know or don't care about their creation order). On the other hand, if you have an important and permanent process (one that lives for as long as the praxis itself) and you think that its priority matters, make sure to create it first (or last). Note that the root process is always

[5] It would be easy to introduce explicit priority schemes to PicOS's scheduler. Such exercises have been made, and they have all proved basically useless and wasteful.

the first created process of the praxis. Also note that it can be terminated, if its role has been fulfilled. That will not automatically terminate the processes started from it.

The amount of memory taken by the PCB of a process (this is the only memory reseource needed to keep a process around) is:

N = F + E x **MAX_EVENTS_PER_TASK**

bytes, where F = 10 for MSP430 and 16 for ARM, E = 4 for MSP430 and 8 for ARM, and MAX_EVENTS_PER_TASK is set to 4 by default (see Section 3.4). This translates into 26 bytes for MSP430 and 48 bytes for ARM.

One consequence (we shouldn't hesitate to say “advantage”) of the synchronous operation of FSM processes in PicOS is the lack of inter-process synchronization problems. All the statements that an activated process executes in its current state are carried out with mutual exclusion with respect to all other processes. Note that this only applies to processes: interrupts (kernel activities) still have to synchronize with processes, but this doesn't concern the application programmer (and doesn't even concern the processes as such).

Consider the operation **proceed** (e.g., see page 5) which provides for a direct transition to the indicated state. To completely understand its behavior, you need to know about scheduling. Although the operation may connote with **goto**, it is important to realize that it is more complicated than that and that it involves the CPU scheduler. In particular, it means that its is possible for another process to grab the CPU during the transition. The operation performs an equivalent of **release** (i.e., a return to the scheduler) with the same process being immediately ready to run in the new state. If a higher priority process happens to be ready, it will be run before the current process gets to executing statements in the new state. Thus, it is not impossible for another process to intervene (run through one of its states) during the transition. This behavior is usually quite intended and even desirable, as it tends to improve the overall responsiveness of the praxis. For illustration consider this code fragment:

```
...
        state MORE_TO_DO:
              if (dt->flag == FG_FINE)
                    trigger (SOME_EVENT);
              proceed TRY_AGAIN;
...
```

Suppose that the **if** condition evaluates to true (**YES**) and the FSM executes **trigger**. The event is then likely to render some other process ready. If the priority of that process happens to be higher than that of the current one, it will be run before the current process gets to executing statements in state **TRY_AGAIN** (wherever its is). The transition is non-trivial and it allows the current process to interleave with others, thus breaking the mutual exclusion.

There may exist situations where one would prefer to use an actual **goto** (which is legal). Here is a convincing (if somewhat artificial) example:

```
fsm one {
        state WAIT:
              when (EVENT1, GOTIT);
              release;
        state GOTIT:
              ...
```


```
                trigger (EVENT2);
                proceed WAIT;
}

fsm two {
        state WAIT:
                when (EVENT2, GOTIT);
                release;
        state GOTIT:
                ...
                trigger (EVENT1);
                proceed WAIT;
}
```

The two FSMs want to play a (symmetric) ping-pong game with signals (events) whereby **one** triggers an event awaited by **two**, which does something in response and replies by triggering an event awaited by **one**, and so on. Suppose that **one** receives the signal and wakes up in state **GOTIT**. Then it triggers **EVENT2** and proceeds back to **WAIT** to await **two**'s response. Suppose that the priority of **two** is higher and whatever additional actions it performs in state **GOTIT** (having received the signal from **one**) involve no blocking. Then, owing to the fact that **proceed** (at **one**) causes the CPU to be reassigned to **two** (which has just become ready), **two** will trigger **EVENT1** before **one** gets to executing **when**. Consequently, the event will be missed by **one**. This is because a triggered event must be awaited (**when** must have been issued for it already) before it can be received (events are not queued). This kind of non-determinism is probably not what the programmer had in mind. The problem will be avoided if **proceed** (in both FSMs) is replaced with **goto**. As **goto**s (and labels) may appear irritating to purists, PicOS offers a somewhat camouflaged **goto** to a state. Here is the relevant fragment of **one** rewritten:

```
        ...
        state GOTIT:
                ...
                trigger (EVENT2);
                sameas WAIT;
        ...
```

The keyword **sameas** (instead of **proceed**) means that you want to execute *the same* code *as* at the indicated state, i.e., as if the code were replicated substituting for the **sameas** statement.

Note that the above example is artificial and meaningless unless the dotted parts (following the state **GOTIT**) contain non-trivial operations (and more states) that (at least in some circumstances) do block.[6] Then, of course, there is an easy way to rewrite the FSMs in a way that eliminates the problem without using **sameas** (and even makes the code a bit shorter):

```
fsm one {
        state GOTIT:
                ...
                trigger (EVENT2);
        initial state WAIT:
                when (EVENT1, GOTIT);
}
```

[6] If this is not the case, the FSMs will lock into an infinite loop.

As we shall see in section 2.6, a process's reaction to an event is usually programmed quite differently, in a way that eliminates the kind of race inherent in the above example. Consequently, you practically never need **sameas**, and if you think you need it, then probably something is wrong with your design.

2.6 Events

Even though many wait & block scenarios (involving wait requests followed by **release**) are buried inside standard functions (and thus are not directly visible to the programmer), the three basic operations needed to implement *all* such scenarios, i.e., **delay**, **when**, and **release**, can be (and often are) invoked by the program directly. With **delay**, a process can set up a timer for a given number of ticks of the internal clock. One such tick corresponds to 1/1024 of a second, which interval is often referred to as the "PicOS millisecond", or just "millisecond", if the context is clear. For example, this FSM:

```
fsm blinker {
      state TURN_ON:
            leds (0, 1);
            delay (512, OFF);
            release;
      state TURN_OFF:
            leds (0, 0);
            delay (768, ON);
}
```

turns LED 0 on and off in 1.25-second cycles with the LED going on for 500 (true) milliseconds and then going off for 725 milliseconds.

A process can issue multiple **delay** requests at a single activation, but only the last of them is going to matter. The maximum value of a **delay** interval is 65535 (the first argument is of type **word**), i.e., 64 seconds minus one tick.

Operation **when** makes it possible to wait for "named" events which in fact are **aword**-sized, unsigned integer numbers. The first argument of **when** can be of any type that is formally cast-able to **aword**, in particular, it can be a pointer. As a rule, it is recommended to use addresses (pointers) for event identifiers. Typically, such identifiers naturally couple with data structures (objects) related to those events and are thus automatically unique. Note that the PicOS kernel, device drivers, and some library functions use the same event signaling mechanism as your FSMs. By consistently following the recommendation you will make sure that your events do not accidentally coincide with those internal events.[7]

Let us try to program a variant of the above led-toggling FSM making it possible to control the blinking from another process. Here is something that one might arrive at via a straightforward extension of the above example:

```
fsm blinker {
      state TURN_ON:
            leds (0, 1);
            delay (512, TURN_OFF);
            when (LED_OFF_EVENT, IDLE);
            release;
```

[7] Such a coincidence, even if it occurs, need not be disastrous (or even detrimental) to the praxis (see below).

```
        state TURN_OFF:
              leds (0, 0);
              delay (768, TURN_ON);
              when (LED_OFF_EVENT, IDLE);
              release;
        initial state IDLE:
              leds (0, 0);
              when (LED_ON_EVENT, TURN_ON);
}
```

The idea is to have two signals that a controlling FSM would send to **blinker** in order to switch it on and off. One extra state, **IDLE**, is the (initial) off state.

Although likely to do its job in a sane situation, the above FSM illustrates how *not* to program such scenarios. The solution suffers from the uncomfortable moments of uncertainty when the process transits among its states (something we already witnessed in the example on page 14). For example, consider the situation when the process has executed **release** in state **TURN_OFF** awaiting both the timer event and **LED_OFF_EVENT**. Suppose that the timer goes off. At that moment the process becomes ready to use the CPU; however, it is not impossible that another process (with a higher priority) also becomes ready at the same time. Depending on the complete picture, this may be in fact impossible, possible but extremely unlikely, or even not so unlikely. For example, the other process may also be waiting for the timer (using **delay**) and may specify the same interval. As the formal activation of a process waiting for the timer (its status change) occurs on a tick of the system clock, all processes with the same target tick will be activated at *exactly* the same time. If something like that happens, and the other process is the controlling process for **blinker**, and it decides to trigger **LED_OFF_EVENT** in its current state, that event will be lost, i.e., it will not be noticed by **blinker**.

A standard and foolproof way to program **blinker**, as well as many other scenarios fitting the same pattern, is to have a clear indication of the status (in some global variable) and only use the signals to indicate/perceive changes in that status, e.g.,

```
Boolean led_is_on = NO;
...
fsm blinker {
        state CHECK_IT:
              if (led_is_on) {
                    leds (0, 1);
                    delay (512, OFF_PERIOD);
              } else
                    leds (0, 0);
              when (LED_EVENT, CHECK_IT);
              release;
        state OFF_PERIOD:
              leds (0, 0);
              delay (768, CHECK_IT);
              when (LED_EVENT, CHECK_IT);
}
```

Then, the operation of switching the blinking on and off can be encapsulated into a function:

```
void blink (Boolean on) {
```


```
        led_is_on = on;
        trigger (LED_EVENT);
}
```

Depending on your taste, the second **when** (at state **OFF_PERIOD**) is optional. In the situation when the blinking is switched off and then quickly on while the process is still waiting on the 2-second delay, that statement will cause the LED to be lit immediately.

Note that the new interpretation of the **when** event is not to tell **blinker** what to do, but rather to bring to its attention the fact that the LED status may have changed (and it is that status that tells the process what to do). This is the most natural and recommended way to treat all named events. Note that the presence of the status variable also immediately answers the question what number to use for the event identifier: clearly, the address of the status variable, i.e., you may want to put this statement:

```
#define LED_EVENT (&led_is_on)
```

somewhere in front of the FSM definition.

As one more leg of this exercise, let us turn **blinker** into a parameterized (strand) FSM capable of controlling any LED, e.g., in this way:

```
typedef struct {
      byte led, state;
} led_status_t;
...
fsm blinker (led_status_t *lstat) {
      state CHECK_IT:
            if (lstat->state) {
                  leds (lstat->led, 1);
                  delay (512, OFF_PERIOD);
            } else {
                  leds (lstat->led, 0);
            }
            when (lstat, CHECK_IT);
            release;
      state OFF_PERIOD:
            leds (lstat->led, 0);
            delay (768, CHECK_IT);
            when (lstat, CHECK_IT);
}
...
void blink (led_status_t *lstat, Boolean on) {
      lstat->state = on;
      trigger (lstat);
}
```

This time we do not shy away from using addresses (pointers) directly as event identifiers (for **when** as well as for **trigger**).

In those rare cases when you want to pass signals between processes, and there is no obvious data structure whose address can be used as the event identifier, you can use the process ID, i.e., the value returned by **runfsm**. Although formally process IDs are integers (of type **aword**), they are in fact addresses of the respective PCBs (cast to **aword**). Thus, for as long as the praxis consistently obeys the principle of associating addresses with all events, process IDs automatically provide unique event identifiers.

If you need an event to be passed to a cohort of processes running the same code (FSM), a natural choice is the address of the FSM function, which you can simply refer to via the FSM name, e.g.,

```
fsm my_fsm {
...
}
        ...
        trigger (my_fsm);
        ...
```

When a process terminates (in whatever way), the two events, i.e., the process ID event + the FSM code event, are triggered automatically by the system (see page 28). There is no harm if nobody is waiting for them. An event that is not expected (there's no pending **when** for it) is simply ignored with no consequences.

Note that overloading events (i.e., using the same event number for several different purposes) is generally OK, as long as the events are interpreted in the recommended way, i.e., as indications of a (possible) status change, with the actual status stored in a variable and verified upon activation. This way, a spurious event will cause no problem: the process will effectively ignore it having found that the status hasn't really changed. The same way, accidental conflicts in event identifiers may bring about some unnecessary process activations, but they will not result in a misbehavior of the praxis.

2.7 External blocking

Blocking can be implemented externally, i.e., in a (regular) function invoked by an FSM. Here is an example:

```
void run_more (word st, word n, aword param) {
        if (crunning (some_fsm) >= n) {
                when (some_fsm, st);
                release;
        }
        runfsm some_fsm (param);
}
```

This function creates a new instance (strand) of **some_fsm** (passing it **param** as the data attribute) if the total number of its instances already running is less than **n**. Otherwise, the process invoking **run_more** will block until at least one of the already running instances of **some_fsm** terminates (in which case the invoking process may want to re-execute **run_more**). Thus, the function accepts a state argument. Here is its sample usage:

```
fsm run_more_caller {
        aword k;
        ...
        state RUN_SOME_MORE:
                run_more (RUN_SOME_MORE, 4, k);
        ...
}
```

Note that whatever statements follow the call to **run_more**, they will only be executed if the function returns. Thus, if the process wants to wait for some other events while it

remains blocked by **run_more** (e.g., a timeout), it must issue the respective requests before calling **run_more**, e.g.,

```
...
state RUN_SOME_MORE:
      delay (4096, GET_LOST);
      run_more (RUN_SOME_MORE, 4, k);
      ...
```

But then, if **run_more** doesn't block, the timer will be set anyway, which may not be what the programmer has ordered. One way out may be to reset the timer to some large interval (note that each **delay** cancels the previous timer setting), but it probably doesn't strike you as an elegant solution. The proper solution is to take advantage of **unwait**, e.g.,

```
...
      state RUN_SOME_MORE:
            delay (4096, GET_LOST);
            run_more (RUN_SOME_MORE, 4, k);
            unwait ();
            ...
```

which cancels all wait requests (**when** as well as **delay**) issued so far in the present activation.

One natural and generic behavior of a blocking function is to try something and either succeed (and return) or find that the goal cannot be accomplished right away. In the latter case, the function will issue a wait request for some event, whose occurrence will be interpreted as a new opportunity to try again, and execute **release**. Thus, when the invoking process is activated in the specified state, it should re-execute the function. An alternative behavior is possible where the activation of the invoking process in the specified state means that the goal has been accomplished (there is no need to re-try anything). A function subscribing to this kind of behavior will return immediately having issued a wait request for some event indicating the completion of its operation. The system functions **delay** and **when** are the most prominent exemplifications of that behavior. By convention, functions falling into the first class accept the state as the first argument, while the other ones expect it at the end of the argument list.

2.8 Example: a complete praxis

Now we shall turn the **blinker** FSM from section 2.6 into a complete praxis. First, we notice that the FSM can be simplified by storing slightly more information in the state attribute of this structure (representing the LED to be controlled):

```
typedef struct {
      byte led, state;
} led_status_t;
```

In the present variant, the values of **lstat->state** are interpreted this way:

0 the LED is off solid
1 the LED is in the “off” phase while blinking
2 the LED is in the “on” phase

```
fsm blinker (led_status_t *lstat) {
      state CHECK_STATUS:
```


```
            if (lstat->state < 2) {
                  leds (lstat->led, 0);
                  if (lstat->state) {
                        lstat->state = 2;
                        delay (512, CHECK_STATUS);
                  }
            } else {
                  leds (lstat->led, 1);
                  lstat->state = 1;
                  delay (768, CHECK_STATUS);
            }
            when (lstat, CHECK_STATUS);
}
```

This new version of **blinker** has only one state. This modification calls for the following revision of the accompanying function:

```
void blink (led_status_t *lstat, Boolean on) {
      lstat->state = on ? 2 : 0;
      trigger (lstat);
}
```

To turn the example into a complete praxis that can be compiled and loaded into a real device (or executed as a VUE[2] model) we need a root FSM. Here is one:

```
fsm root {
      led_status_t *my_led;
      state START:
            ser_out (START,
                  "Commands:\r\n"
                  "  on\r\n"
                  "  off\r\n"
            );
            my_led = (led_status_t*)
                  umalloc (sizeof (led_status_t));
            my_led -> led = 1;
            blink (my_led, NO);
            runfsm blinker (my_led);
      state INPUT:
            char cmd [4];
            ser_in (INPUT, cmd, 4);
            if (cmd [1] == 'n')
                  blink (my_led, YES);
            else if (cmd [1] == 'f')
                  blink (my_led, NO);
            else
                  proceed BAD_INPUT;
      state OKMSG:
            ser_out (OKMSG, "OK\r\n");
            proceed INPUT;
      state BAD_INPUT:
            ser_out (BAD_INPUT, "Illegal command\r\n");
            proceed INPUT;
}
```


The process starts in state **START** where it writes to the UART a message presenting two simple commands: “on” and “off.” Then it allocates memory for the LED data structure and sets the LED number to 1. Next it calls **blink** to turn the LED initially off (the standard symbolic constant **NO** is equivalent to 0, i.e., *false* – see page 8). Note that the first event triggered by **blink** will be ignored as nobody is expecting it yet (the **blinker** process gets created by **root** after the call to **blink**).

In state **INPUT** the process tries to read from the UART (function **ser_in**) a line storing up to four initial characters of that line in array **cmd**. The second character of the read sequence is used to tell which command it is; then a pertinent call to **blink** is issued.

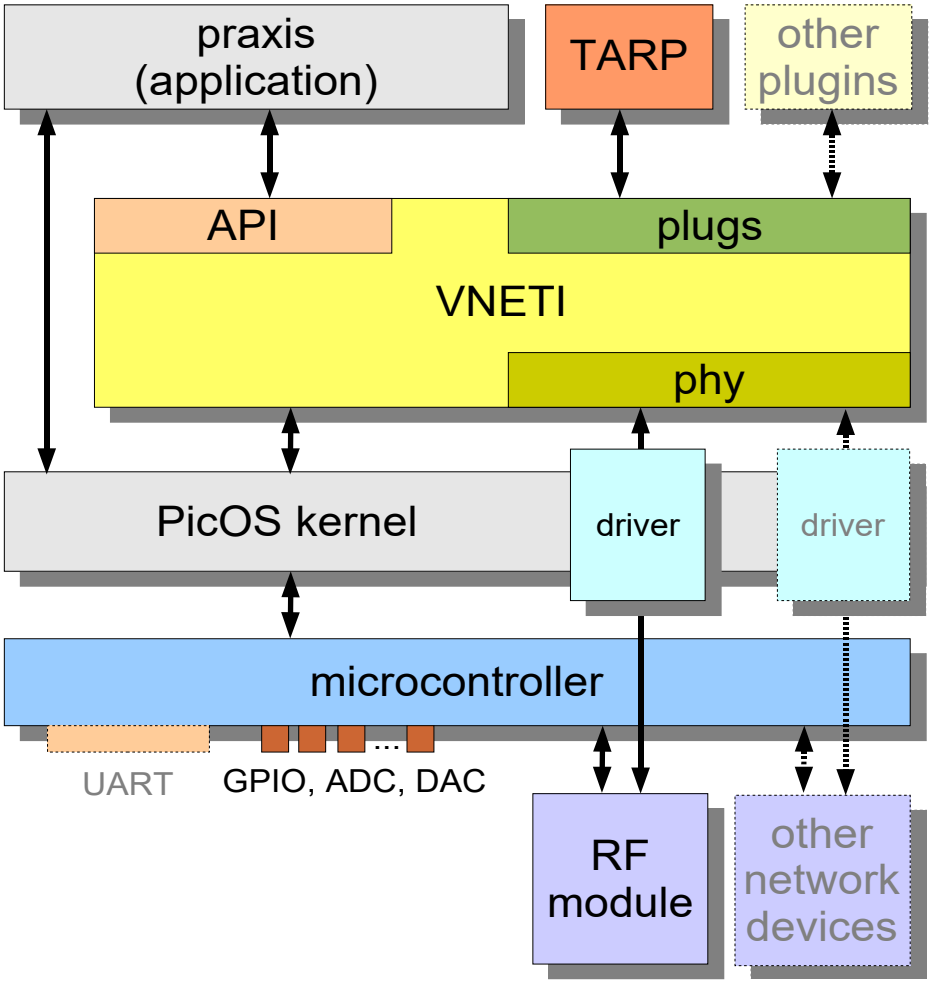

Figure 1: System structure.

Note that the **cmd** array (passed as the second argument to **ser_in**) is state-local (i.e., allocated on the stack). In confrontation with the fact that function **ser_in**, responsible for filling that array with data, can block, this brings about some worries. The intriguing question is how does **ser_in** fill **cmd** with data? Is it possible, for example, that the function stores some (immediately available) data first and then blocks expecting to store more on subsequent invocations? Or perhaps ser_in supplies the address of **cmd** to some (interrupt driven) driver that will be filling the array “off-line” (while the process is blocked). That would not work, as we cannot reasonably assume that any data stored in

cmd will survive blocking (and re-entering the state). Moreover, while the present process remains blocked, the address of **cmd** (which formally doesn't exist at the moment) is likely to be used by another process.

In order to use a local array in this context we have to *know* that **ser_in** handles it safely (i.e., doesn't use it as temporary storage). Indeed, this is how the function works: the data are only stored in the array, if a complete line is immediately available in some internal buffer. We say that this (return) argument of **ser_in** is blocking-safe. Clearly, it makes sense to try to make all arguments of potentially blocking functions blocking-safe, if only possible. Generally, this issue is something that you should be aware of, especially when implementing your own blocking functions.

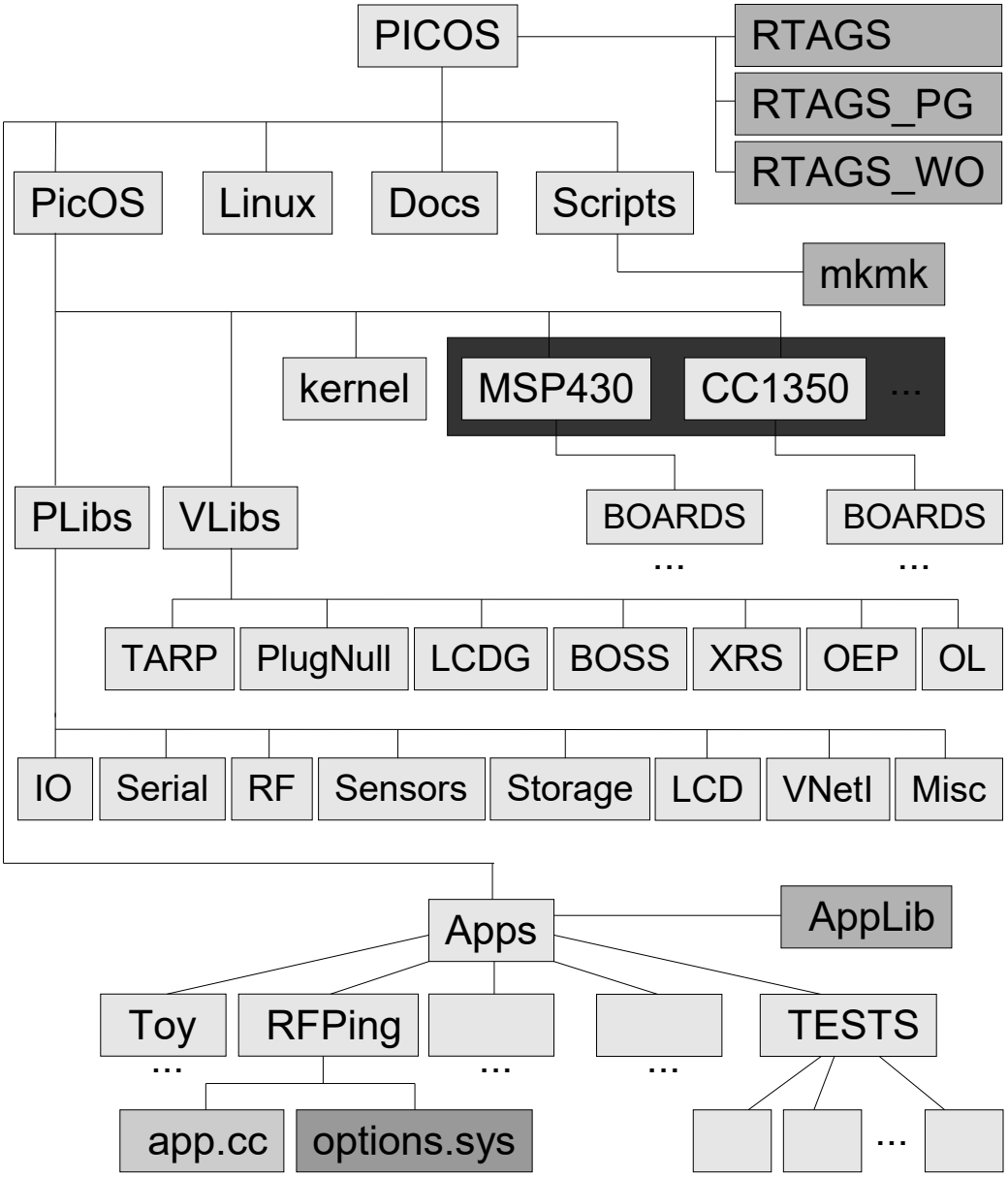

Figure 2. PicOS directory tree.

3 System organization and structure

The PicOS platform consists of four components: PICOS, PIP, SIDE, and VUEE, which come as four separate packages. Formally, PICOS alone is sufficient for preparing programs for target (real) microcontroller devices. PIP offers a GUI-based SDK to assist the user in that task, while SIDE and VUEE together provide a virtual execution environment for PicOS praxes making it possible to develop them in isolation from real hardware.

The procedure for setting up the platform is described in a separate document [installation]. Here we only focus on one component of the platform, i.e., the PICOS package.

3.1 Logical view of the system

PicOS can talk to many types of peripheral devices, e.g., UARTs and RF modules. Usually, an RF module (and, in many cases, also the UART) is interfaced to the praxis via VNETI [vneti], which brings in an orthogonal, simple, and powerful collection of tools for describing possibly complex I/O sessions (protocols) involving packets exchanged by the node with the various devices. The root functionality of VNETI is extensible via *plugins*. This way, e.g., networking in PicOS is layer-less. Figure 1 shows the logical organization and interactions among the various system components.

A PicOS program is always intended for a specific *board*. The system includes an extensible set of board descriptions (for each generic CPU type) specifying details like I/O pin configurations, RF module interface, UART parameters, etc., which depend on the layout of a particular device. New board descriptions can be created relatively easily by cloning and modifying the existing ones.

3.2 Package content

Figure 2 shows the layout of directories of the PICOS package. Files whose names begin with *RTAGS* (located at level zero) contain the history of modifications along with the tags identifying the respective versions of the system. The main-level directories store the following components:

- *Apps*

  This is a hierarchical repository for praxes, with each praxis occupying one leaf-level sub-directory. At least one file is required to describe an application: *app.cc* 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.

  In the past, until version PG170720A, all praxes must have been stored under *Apps,* to be accessible by PicOS tools. These days a praxis can be kept anywhere within the file system hierarchy, and it actually makes better sense to store it separate from the PICOS tree. Thus, the *Apps* directory that comes with PICOS contains only some standard (historical or illustrative) collection of praxes.

  The role of the *AppLib* subdirectory of *Apps* is to contain the so-called application library, i.e., source files and headers that can be shared by different applications. This is similar to to the *VLibs* subdirectory of *PicOS* (see below), except that the latter is intended for system modules while the former is extensible by the user.

- *PicOS*

This directory contains the source code of the system kernel, standard library modules, device drivers, etc. In addition to a few C files containing various architecture-independent components of the system, the directory includes several subdirectories:

kernel containing the architecture-independent kernel code

CC1350 containing the components pertinent to the CC1350 CPU (including architecture-specific parts of device drivers)

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

PLibs containing CPU-architecture-independent system modules partitioned into these subdirectories:

- *RF* radio device drivers
- *IO* drivers for other (non-RF) devices
- *Sensors* drivers and low-level system code for handling sensors and actuators
- *Serial* architecture-independent UART drivers, protocols, and application-level interface
- *LCD* drivers for LCD devices
- *Storage* drivers for EEPROM and flash devices, including SD cards
- *VNetI* VNETI interface functions
- *Misc* other device drivers and system modules, e.g., RTC (real-time-clock) drivers, encryption and checksum functions

VLibs containing higher-level hardware-independent system components, ones that can be directly used in VUE[2] models[8]) partitioned into these subdirectories:

- *TARP* the TARP (ad-hoc) forwarding scheme (a VNETI plugin)
- *PlugNull* the NULL plugin to VNETI shorting the interface for raw (direct) communication
- *XRS* the protocol suite of the so-called External Reliable Scheme for reliable exchange of data over UART and wireless links
- *OEP* the Object Exchange Protocol suite, i.e., tools for reliably transmitting large objects over low-bandwidth unreliable links [oep]
- *OL* the Object Loader, i.e., a simple tool for loading data structures from EEPROM or flash in a relocatable way [eol]
- *LCDG* tools for creating menus and hierarchies of displayable objects on tiny graphic LCDs, e.g., Nokia N6100 [lcdg]
- *BOSS* the BOSS plugin for implementing reliable OSS interfaces over UART [oss]

[8] When the praxis is compiled for VUE[2], these components are directly compiled [picomp] into the model.

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

- *Linux*

  This (legacy) directory contains miscellaneous programs developed under Linux including a prototype location tracker.

- *Docs*

  Documentation directory.

- *Scripts*

  Scripts mostly in Tcl. The most important of them are:

  mkmk the PicOS maker generating SDK projects and makefiles for PicOS praxes [mkmk]

  picomp the PicOS compiler transforming source files with PicOS programs into straightforward C code; also used to transform PicOS praxes into VUE^2 models [picomp]

  piter a cross-platform terminal emulator compatible with PicOS-specific UART communication modes [serial]

3.3 Board descriptions

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

3.4 System parameters

There are two basic ways of building a praxis image that can be uploaded into a hardware node: the so-called source build (with the PicOS source recompiled for the praxis), and the the library build (using a precompiled object library, which is typically board-specific). See [mkmk] and [pip] for details.

Most of the system configuration options (those of a general nature) are listed in file *modsyms.h* in *PicOS* as **#define** constants together with explanations. The following two constants determine important limits of which the praxis may want to be aware:

MAX_EVENTS_PER_TASK (default value: 4)

The maximum number of different events that a single process can await simultaneously, excluding the timer (**delay**) event.

STACK_SIZE (default value: 256)

The amount of RAM set aside for the stack (in bytes).

It is possible for the praxis to monitor the stack size and diagnose its overflow.[9] Here is one more parameter to this end:

STACK_GUARD (default value: 0)

This is a Boolean value with nonzero selecting the stack guard option, where the system (heuristically) monitors the stack against overflows. The stack is pre-filled with words of a certain pattern, which allows the system to:

1. determine the maximum usage of the stack so far
2. raise an alarm (and possibly reset the CPU) when the last location of the stack has been written to

With **STACK_GUARD** != 0, the system function **stackfree** (see page 41) returns the minimum number of free stack locations observed so far (in **awords**).

One more global parameter of a potential interest to the praxis is:

MAX_TASKS (default value: 0)

which selects the process ordering option discussed on page 13. In previous versions of the system, the constant (also) used to limit the maximum number of processes in the system (hence the confusing name), but at present it is treated as a binary flag: zero means that new processes are appended at the end of the process list, nonzero means that they are inserted from the front.

4 Essential operations provided by the kernel

Below, we list the essential operations that the system makes available to its processes, in the loose order of their importance. Note that there are three types of operations:

1. C functions as well as macros mapping into straightforward function calls (in the list below they are marked with **F** on the right). For such an operation, we list the function header specifying the argument types as well as the type of the return value.
2. Function-like macros that cast (possibly some of) their arguments. These operations are marked with **M**. While retaining the function syntax, such a macro accepts arguments of flexible types. For an argument of this kind, we will put the

[9] There has not been a single (noted) case of stack overflow in the entire history of our software development with PicOS. I want to contrast this to TinyOS where stack overflow is a notorious problem.

+ sign in front of the argument type, meaning that whatever you specify as the actual argument must be cast-able to the listed type.

3. Operations not following the C function syntax (marked with o). The arguments (and the result) of such an operation are described individually in each case.

```
runfsm fsm_name                                                    o
runfsm fsm_name (data)                                             o
```

This operation creates a new process and returns its identifier (PID) of type **aword**. This means that the construct can occur on the right hand side of an assignment (and also in an expression) even though it doesn't look like a function call. *fsm_name* is the name of an FSM function that must have been defined or declared (announced) earlier (section 2.3). The variant with parentheses and argument is used to create a strand (section 2.3). In such a case, the argument must be of a simple type, i.e., something that can be cast to **aword**. The argument type is *not* checked for agreement with the data type specified with the FSM definition.

The only reason why **runfsm** may fail is the lack of memory for the process's PCB. The operation will fail if **umalloc** for the PCB returns NULL (see page 40).

When the process executing **runfsm** finds that it has failed, it may want to wait for an opportunity to try again. Whenever a process is terminated (and a PCB slot becomes free) the system will trigger a pool-0 memory event,[10] which can be awaited with **waitmem** (see page 40), also using this alias:

```
void npwait (word st)                                              F
```

where **st** is the state where the process wants to be resumed when memory (that can be used for a PCB slot) becomes free.

```
void kill (aword pid)                                              F
```

This function kills the indicated process removing it from the system and freeing its PCB. As a special case, **kill(0)** kills the current (invoking) process and is exactly equivalent to:

```
finish                                                             o
```

Note that neither **finish**, nor **kill (0)** ever returns.

Killing/finishing a process triggers two events: the first one is numerically equal to the PID of the terminated process, the second is equal to the address of its FSM function. For example, to wait for the termination of any process running a specific FSM, you should issue **when** specifying the FSM function pointer as the first argument. There exist two aliases for such occasions:

```
aword join (aword pid, word st)                                    F
```

The operation issues a wait request for the termination of the process identified by the first argument (PID). The function returns the first argument (PID) if it identifies a legitimate process (and the wait request has been in fact issued), or 0, if no such

[10] These days the event is strictly equivalent to “some memory in pool 0 becoming free” (which is covered by **waitmem**). There is an obscure historical reason for the alias.

process exists (no wait request has been issued then). Note that it is often tempting to use a simple construct like this:

```
join (runfsm some_fsm (some_arg), some_state);
```

the problem being that (in principle) **runfsm** can fail on memory allocation for the new process. In such a case, the first argument will amount to 0 and **join** will return 0 indicating that it has been ineffective.

void joinall (fsmcode fsm, st) **F**

This is equivalent to:

```
when (fsm, st)
```

The operation issues a wait request for the termination of any process running the code of the specified FSM.

void killall (fsmcode fsm) **F**

This function kills all process instances running the specified FSM function. It is OK if the operation finds no such process (it will then do nothing). The effect is exactly as if **kill** were executed in turn for every qualifying process.

aword running (fsmcode fsm) **F**
word crunning (fsmcode fsm) **F**

If an instance of a process running the specified FSM function is present in the system, **running** returns the PID of one (any) such an instance.[11] Otherwise, the function returns zero. The second function (**crunning**) returns the number of processes that run the specified FSM (see page 19 for an illustration).

If the argument of **running** is **NULL**, then the function will return the PID of the current process (the one that has invoked the function). Here is an alias for such a call:

aword getcpid () **F**

fsmcode getcode (aword pid) **F**

Returns a pointer to the FSM function executed by the indicated process (and can be viewed as a counterpart of **running**). If **pid** is zero, the pointer to the current process's FSM function is returned. This may make sense when the operation is issued from a function external to the process's FSM code.

void when (+aword ev, word st) **M**

Issues a wait request for the indicated event (**ev**). When the event is triggered, the process will be resumed in the specified state (**st**). Although **st** 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 because it conflicts with VUE[2].

[11] In fact, this is the highest-priority process among them (see section 2.5).

Note that the first argument of **when** can be anything cast-able to **aword**.

void trigger (+aword ev) **M**

Triggers the indicated event (that can be awaited by **when**).

For the purpose of **when** and **trigger**, events are identified by **aword**-type numbers (with different values treated as different events). Certain internal or special-purpose events can be awaited (implicitly) via some other operations (e.g., **waitmem**). As we postulated in section 2.6, the praxis should use addresses as event identifiers (all events used by the kernel and library modules follow this postulate). There is no need to cast such values (to **aword**) when putting them as arguments of **when** or **trigger**.

Triggered signals are never queued (they are no objects): if no process is waiting for a triggered event, the **trigger** operation is completely void. On the other hand, if there are multiple processes awaiting for the same event delivered by **trigger**, all of them will be awakened (turned ready) simultaneously.

void ptrigger (aword pid, +word ev) **M**

This operation acts exactly like **trigger** (see above), except that the event is only presented to the indicated process. If there is only one process interested in receiving the event and its PID is known, **ptrigger** is a bit more efficient than **trigger**. The **pid** argument must be a valid process ID and, in particular, it cannot be zero.

release **O**

This parameter-less operation leaves the current process, i.e., forces the completion of the current state. When executed from within the process's FSM function, it can be replaced by **return** (although doing so is not recommended from the viewpoint of clarity). Note that **release** can be executed from a function called from the FSM, in which case it will exit the process rather than return from the function.

void delay (word timeout, word st) **F**

This operation issues a timer wait request, i.e., sets up an alarm clock to go off after the specified number of *PicOS milliseconds.*[12] When that happens, the process will be activated 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 becomes effective. A **delay** request is interpreted in a way similar to a **when** request and can naturally coexist with other (simultaneous) wait requests.

word dleft (aword pid) **F**

This function returns the residual number of PicOS milliseconds for which the indicated process will continue to sleep on the **delay** timer (see above), i.e., remaining until the timer goes off (assuming that 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_WORD**. The case **pid** == 0 is interpreted as an inquiry about the current process. In such a case, it will return the

[12] One PicOS millisecond equals 1/1024 of a second.

setting of the last **delay** issued by the process in its present state, or garbage, if the process has not issued a **delay** request in its current activation.

lword seconds (void) **F**

This function returns the number of true seconds elapsed from the moment the system was started (reset).

proceed st **O**

This operation unconditionally transits to the indicated state **st**. The transition is different from a **goto** in that it involves the CPU scheduler (as explained in section 2.5). 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.

sameas st **O**

This operation unconditionally transits to the indicated state **st** in a manner similar to **goto**. The effect is exactly as if the FSM executed **goto** to a label declared before the first statement of the specified state.

Note that both **proceed** and **sameas** require a symbolic state identifier, which must appear at one of the **state** statements of the FSM. This is in contrast to functions expecting a state identifier, which treat the state argument as a number. Such a number can be stored in a variable (or even calculated) and passed from that variable (or from an expression).

Consequently, you can only execute **proceed**/**sameas** from an FSM function (you cannot do that from a function that has been invoked from the FSM). While **sameas** is truly impossible in such a context, **proceed** can be emulated as:

```
delay (0, st);
release;
```

To make it absolutely foolproof (and make sure that no event requested earlier by the process will interfere with the transition), you may precede **delay** with:

void unwait () **F**

which operation revokes all wait requests (**when**, **delay**, including their aliases) issued by the process so far in its present activation.

call *fsm_name* (+aword data, st) **O**
call *fsm_name* (st) **O**

This operation creates a process from the indicated FSM and suspends the caller until the new process terminates. Then, the invoking process will be resumed in the specified state (**st**). It is formally equivalent to this sequence:

```
{
        aword pid;
        pid = runfsm fsm_name (data);
        if (pid != 0) {
                join (pid, st);
```


```
                release;
            }
        }
```

with the **(data)** part absent in the second variant. Thus, the operation does not return, unless the process could not be created (because of the lack of memory for a PCB slot).

Here is a sample sequence to issue the call reliably, i.e., to persist until the called process has been in fact created:

```
        ...
    state CALL_IT:
        call my_fsm (my_data, CALL_DONE);
        npwait (CALL_IT);
        release;
    state CALL_DONE:
        ...
```

void savedata (+aword data) **M**

The data attribute (see pages 5 and 8) associated with the current process (strand) is set to the specified value. Normally, the data attribute available to the strand is not expected to change during the process's lifetime. Even though the process can formally change the data attribute (which appears to it as a local variable), the change is undone after every state transition, because whenever the process is activated, the attribute is re-loaded from its original value stored in the PCB. With **savedata**, the process can reset the hard copy of the data attribute in the PCB.

For illustration, consider the **toggle** FSM from Section 2.4 which stores the on/off status of the LED in a trivial structure pointed to by the process's data attribute. With **savedata**, the status flag can be stored directly in the data attribute, e.g.,

```
    fsm toggle (byte on) {
        state TOGGLE_LEDS:
            leds (0, on); leds (1, on = 1 - on);
            savedata (on);
        initial state WAIT_FOR_TIMER:
            delay (2048, TOGGLE_LEDS);
    }
```

void utimer_add (word *utm) **F**

This operation declares a countdown timer. The first argument points to a **word** variable which will be added to the global pool of countdown timers. As soon as its contents are set to a nonzero value (with **utimer_set** – see below), the word will be decremented towards 0 at millisecond[13] intervals. The maximum size of the utimer pool is 4 entries (an attempt to create more than 4 simultaneous utimers will trigger a system error).

void utimer_set (word utm, word val) **M**

[13] PicOS milliseconds, i.e., 1/1024 s.

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

A utimer operates in the background independently of the activities of the process that has set it up. In contrast to a **delay** timer, it doesn't trigger any events, but offers a running readout that a process can consult while performing other operations. For illustration, suppose that a certain FSM responds to two types of conditions/events: one of them to be handled immediately upon occurrence and the other debounced at some intervals (i.e., its close repetitive occurrences should be ignored). Here is a skeletal layout of such an FSM:

```
fsm handler {
      word debounce = 0;
      ...
      state INIT:
            utimer_add (&debounce);
            ...
      state LOOP:
            if (cond_one) {
                  … handle condition one …
                  cond_one = NO;
            }
            if (!debounce && cond_two) {
                  … handle condition two …
                  cond_two = NO;
                  utimer_set (debounce, DEBOUNCE_INT);
            }
            when (&cond_one, LOOP);
            when (&cond_two, LOOP);
            ...
```

void utimer_delete (word *utm) **F**

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

void udelay (word del) **F**

This is a CPU-spin delay loop that takes approximately **del** microseconds. Not recommended in serious operations, but handy in tests.

void mdelay (word del) **F**

This is another CPU-spin delay loop that takes approximately **del** milliseconds.

Note that tying up the CPU in a process – for the mere purpose of implementing a delay – is seldom a good idea.

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


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 written);

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

```
void memcpy (+char *dest, +const char *src, sint count)     M
void memset (+char *dest, +char val, sint count)            M
```

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**. Note that the operations are macros with flexible arguments. This alias:

```
void bzero (+char *dest, sint count)                        M
```

is equivalent to:

```
memset (dest, 0, count)
```

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

This function is intended for debugging. It writes a line of text directly to UART A (the first UART, in case there's more than one) 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

Each of the first three format descriptors can be preceded by **l** (e.g., **%lu**) with the effect of extending the corresponding argument to 32 bits.

```
void leds (+sint which, +sint how)                          M
```

This operation 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 one specifies the action, which can be:

0 the LED is turned off
1 the LED is turned on
2 the LED is put into a blinking mode

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

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 leds_all (+sint how) **M**

This operation is the same as **leds** applied to all LEDs at the same time.

void reset (void) **F**

The microcontroller is unconditionally reset and the system is restarted. The function never returns.

void halt (void) **F**

The microcontroller is stopped. The function never returns.

void syserror (sint code, const char *msg) **M**

The function aborts the system presenting the specified error **code**. If the **DIAG_MESSAGES** system option is set to a value greater than 1 (which means that debug messages are verbose), the specified message (**msg**) will be written to the UART. Otherwise, (**DIAG_MESSAGES** is less than 2) the message is ignored (the message string is not compiled in and uses no memory).

void sysassert (+bool cond, const char *msg) **M**

If **DIAG_MESSAGES** is greater than 1, this macro tests the condition specified as the first argument, and if the condition fails, executes **syserror (EASSERT, msg)**. **EASSERT** is defined as 10.[14] If **DIAG_MESSAGES** is less than 2, the macro is void.

word rnd (void) **F**

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, 16-bit) version is selected, which should still be OK for typical applications (e.g., randomized backoff for RF channel acquisition). With **RANDOM_NUMBER_GENERATOR** set to 2, the more advanced (32-bit) version is selected, which produces statistically better results at the cost of increased processing time and the overhead of 2 extra bytes of RAM.

[14] The full list of errors generated by the system can be found in *sysio.h*.

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).

lword lrnd (void) F

This function returns **lword**-size random numbers between 0 and 42949667295. It also works when **RANDOM_NUMBER_GENERATOR** is 1, with the result produced by concatenating two consecutive values returned by **rnd**.

5 The UART

The UART interface described in this section is on the edge of being considered a legacy feature. There exist alternatives, described in [serial], which are recommended for all new praxes.

Note for version 5.0: the "legacy" feature doesn't want to go away, so I think we shall keep it as a compilation option.

5.1 The direct device concept

The first production-level incarnation of PicOS assumed two types of I/O devices:

1. Devices accessed through the PHY interface of VNETI (see [vneti]) with the intention of providing a network-like, packetized, buffered view from the praxis.

2. Devices accessed in a more direct fashion and visible by the praxis via traditional read/write operations. This interface is referred to as **io**.

Over time, the need for the **io** interface has practically disappeared, except for the UART. Even for the UART, its VNETI (PHY) interface [serial] is more flexible and (arguably) no more complex than the **io** interface. However, as the cost of keeping the **io** interface around is low, there is no compelling need to eliminate it, especially that:

1. Old praxes use it heavily.

2. It is arguably more convenient for quick hacks, tests, and debugging. Or, perhaps, it only seems so because of our personal addictions.

In general terms, the **io** interface (to any device) assumes that devices are identified with integer numbers starting from zero. All I/O operations with such a device are performed via this function:

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

In the history of PicOS, there have been exactly three devices interfaced this way: the UART, the LCD from the old eCOG1 evaluation board by Cyan Technologies, and the ETHERNET chip from the same board. Considering that 1) the eCOG1 CPU is obsolete and its support has been retired from PicOS, 2) the LCD interface was unnecessarily clumsy and cumbersome, 3) the ETHERNET interface was considered temporary (to be replaced with a VNETI PHY), it appears that the UART has been the only serious subscriber to the **io** interface.

The five arguments of **io** stand, respectively, for the blocked state identifier, the device number, the operation code, 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**. Three types of operations are available: **READ**, **WRITE**, and **CONTROL**, the last one representing special actions not directly related to data transfers.

Depending on the circumstances, an 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 issuing process is blocked 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 (for example, UART **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) wait requests issued by the process. Thus, if the process simultaneously awaits another event, and that event occurs earlier than the status change on 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:

```
...
state SOME_STATE:
      delay (0, WOULDBLOCK);
      ptr += io (SOME_STATE, UART_A, WRITE, ptr, len);
      ...
state 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 result is to use **WNONE** 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 in a significant number of **io** requests the state argument is irrelevant (and thus redundant), the system defines this macro:

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

as a shortcut for such occasions. It expands into a call to **io** with the first argument set to **WNONE**.

5.2 The `io` interface to 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`**). **`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`**, the praxis can issue this (never-blocking) **`CONTROL`** operation to change the UART's baud rate:[15]

```
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 capabilities of the oscillator 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-frequency oscillator, these additional rates become available: 14400, 19200, 28800, 38400, 76800, 115200. 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 (see file *modysms.h* in directory *PicOS*):

```
#define UART_DRIVER                 0
```

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

```
#define UART_RATE                   9600
```

The bit rate for the UART. If two UARTs are present, the rate is the same for both of them. However, if **`UART_RATE_SETTABLE`** (see below) 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 any nonzero value), 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_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 (the 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 acts as a damping storage for

[15] At present, the UART rate on eCOG1 cannot be changed (it is "hardwired" at 9600 bps). Thus, this fragment applies to MSP430 only.

data bursts. Of course, the buffer occupies memory, so it shouldn't be used unless needed. It is generally not needed for keyboard input.[16]

```
#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 a PHY [serial]. This requires **UART_DRIVER** to be 0.

6 Memory allocation

PicOS is equipped with a simple and effective **malloc**-like memory allocator, which makes it possible to organize the dynamically available RAM (known as the heap) into a number of disjoint and non-interfering allocation pools.

Multiple memory pools are enabled by default.[17] In tight memory scenarios, it may make sense to disable them by setting:

```
#define MALLOC_SINGLEPOOL           1
```

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

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

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

which is declarative and should appear in the data section of the program. This statement is not needed (and if present is void) when multiple memory pools are disabled. The specified integer values ***p0, p1, ..., pn-1*** must be all strictly positive and they should add to 100. Their number ***n*** determines the number of memory pools, with each pool receiving the specified percentage of available memory. Pool 0 is intended as the primary storage used by the praxis. PCBs (Process Control Blocks) are also allocated from pool 0. 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 not use pool 1, if VNETI is present.

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

```
address malloc (word pool, word size)
void free (word 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. The allocated chunk always consists of an entire number of **awords** and is aligned at an **aword** 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.

Function **free** accepts **NULL** as the second argument, in which case the operation is void.

[16] Some architectures, e.g., CC1350, provide hardware buffers for the UART.
[17] While formally this is true, all MSP430 boards redefine this parameter opting for a single memory pool. Besides, multiple memory pools are (at present) incompatible with VUE[2].

If multiple pools are disabled, the above functions are compiled 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 (in bytes) of a chunk allocated by **malloc**.

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 ([word 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 minfree)
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 available for allocation in **awords** (not in bytes!). If the second argument of **memfree** is not **NULL**, it is interpreted as the address of the **word**-sized location at which the function will store the minimum observed amount of free memory (also expressed in **awords**). The tracked minimum is then reset to the current amount of free memory (i.e., the value returned by the function). The tracked minimum becomes zero (regardless of the actual amount of free memory formally available) when an allocation request cannot be fulfilled (**malloc** returns **NULL**). 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 memory happens to be fragmented. The second function returns the maximum size (also in **awords**) of a single memory chunk available for allocation. If **nchunks** is not **NULL**, the function will store at the indicated (**word**) location the number of chunks (which can be viewed as a measure of fragmentation).

Note that all sizes returned by the above functions are **word**-type numbers. Theoretically, on some architectures, this may not be enough to accommodate a value. This will happen, e.g., on an ARM device when the amount of allocatable RAM exceeds 256KB (4 x 64K). In such a case, a size exceeding the limit of the 16-bit unsigned number is shown as 65535 (**MAX_WORD**).

If PicOS has been configured with **MALLOC_SAFE** != 0, it will perform some rudimentary consistency tests aimed at catching multiple deallocations of the same block, or a corruption of the free list.

If the system has been configured with **STACK_GUARD** != 0 (see page 27), this function becomes available to the praxis:

```
word stackfree (void)
```

which returns the number of unused stack locations (in **awords**), i.e., the number of stack **awords** that have not been touched by the praxis so far. The measure is obtained heuristically by finding the furthest stack location that does not contain the initialization pattern (see page 27).

7 Power saving tools

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

For MSP430, if the primary crystal is in fact low-speed, the situation is very straightforward. PicOS is then compiled (by default) with the so-called **TRIPLE_CLOCK** option, which provides for an efficient implementation of system clocks facilitating deep power savings (the LPM3 mode of the CPU). For CC1350, two “active” low-power modes are available, dubbed *idle* and *standby*, with the latter being the long-life, deep, still-active, sleep mode corresponding to LPM3 for MSP430. The third low power mode for CC1350, called shutdown, means in fact *shutdown* of the CPU with the I/O pins locked in their last state. The only way to exit that mode is via a state change in one of the selected input pin(s) which will cause a full reset. The process requesting mode 3 will continue, but at the nearest moment when the CPU scheduler finds no ready process (and decides to wait for interrupts), the device will shut down.

This function allows the praxis to select the power mode:

```
void setpowermode (word mode)
```

where the available/useful range of **mode** depends on the architecture. In all cases, zero means full power (the default mode assumed after reset). For MSP430, any nonzero value means LPM3, i.e., the only (deepest active) low power mode available.

For CC1350, **mode** can be 0, 1 (idle), 2 (standby), or 3 (shutdown). Two aliases are available:

```
#define powerup()       setpowermode (0)
#define powerdown()     setpowermode (DEFAULT_PD_MODE)
```

where **DEFAULT_PD_MODE** is defined as 1 for MSP430 and 2 for CC1350. Thus, **powerdown** selects the deepest sleep state during the periods of CPU inactivity that still makes it possible for the CPU to wake up on normal events. Note that mode 3 (shutdown) for CC1350 means in fact shutdown of the CPU with the I/O pins locked in their last state.

While in the **powerdown** mode, the praxis can basically function in the same way as in **powerup**, except that the event (including timer) wakeup time may be slightly increased. All system clocks will function normally. On CC1350, UART requires full power (0) or idle mode (1) to be operable, i.e., it doesn't work in mode 2 (and obviously not in mode 3).

On MSP430, if the primary CPU crystal is high-speed, **TRIPLE_CLOCK** is forced off and then **powerdown** will bring little reduction in power consumption, although it still selects the LPM3 mode for inactive CPU. In that case, the implementation of system clocks is also less efficient.

As a (not recommended) legacy option, **TRIPLE_CLOCK** can be disabled at compilation (also when the primary crystal is low-speed). If **TRIPLE_CLOCK** is 0 and the primary crystal is low-speed, then two more operations become available to the praxis:

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

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

Note that, although **clockdown** and **clockup** are formally available in all circumstances, they are only non-void when 1) the primary CPU crystal is low-speed (at 32768 Hz), **and** 2) the **TRIPLE_CLOCK** option has been **explicitly** disabled at compilation. In all the remaining circumstances, the operations expand into empty statements. The option makes no sense (and is not available) for CC1350.

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

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

void freeze (word nsec)

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

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

Note that normally, when **TRIPLE_CLOCK** is 1, there is no more need for **freeze**. Executing long delays (page 30) or holds (page 46), with no activities in the praxis, amounts to the same thing (except that the seconds clock is advanced normally).

8 The watchdog

The praxis has access to simple watchdog operations listed below (to be implemented for CC1350):

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

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

9 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. Refer to other documents 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 (the default),[18] **'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.

The following five functions (with names starting with "**ser_**") are only applicable if the UART(s) has/have been configured in the **io** mode (see section 5.2).

```
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)
```

[18] It makes sense to switch this option off on tight-memory 16-bit systems, e.g., MSP430. On a 32-bit system, the option is in fact desperately needed, although it can be switched off in principle.

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, 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 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 function can only return when the request has been processed. Then it returns the number of characters stored in the buffer (not counting the **NULL** byte).

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**. It returns the number of items that have been located and extracted from the input line.

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

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

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

Requires *hold.h*. The function implements a long delay until the node's seconds clock (the value returned by **seconds ()** – see page 31) reaches the specified value. The argument must be global, i.e., its value has to survive the process's state transitions, as the function may retry at the indicated state before returning (when the specified second has been reached). Here is an illustration how the function should be called:

```
fsm some_fsm {
        lword del;
        ...
        state INIT_HOLD:
                del = 3600 + seconds ();
                ...
        state HOLD_ME:
                hold (HOLD_ME, del);
                ... the delay has elapsed ...
        ...
}
```

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