Olsonet Communications

VUE^2

Version 0.86

April 2010

1 Introduction

VUE^2 or (VUEE) which stands for Virtual Underlay Execution Engine, is an emulator for PicOS applications and their underlying networks. VUE^2 is facilitated by the fact that PicOS is a relatively straightforward descendant of a powerful simulator named SMURPH. Thus, the Emulation Engine comprising VUE^2 is built around SMURPH, with a natural mapping of many PicOS operations to their SMURPH counterparts.

This document is a constantly evolving draft of VUE^2 description, including its interfaces needed by a PicOS developer to maintain compatibility with VUE^2 (such that the PicOS praxis can be **directly** run under VUE^2). As the system is being developed, this description will change. Note that VUE^2 is modified not only in the course of its natural evolution (as a system on its own), but also in response to new features being added to PicOS (which have to be mirrored in VUE^2). At present, VUE^2 captures a significant subset of all PicOS features, including drivers for RF modules, pin/sensor interface, external memories, UARTs, the Nokia LCD display, and more. Most importantly, it provides for an easy expression of network models and their virtual deployment. This means that PicOS praxes[1] can be run under VUE^2 on a multitude of virtual nodes behaving as if they were interconnected via realistic wireless links. Those virtual nodes can interface to real-life OSS agents in exactly the same manner as real nodes would. Thus, in addition to providing insights into the behavior of a PicOS praxis in the real world (performance assessment, network planning, also including power budget), VUE^2 also facilitates the development of various agents that have to be interfaced to that praxis to make the overall system complete.

This document will not be comprehended without the following accompanying documents by Olsonet Communications: *PicOS, version 2.03* [or higher], *VNETI: Versatile NETwork Interface, version 2.01* [or higher], *SIDE/SMURPH: a Modeling Environment for Reactive Telecommunication Systems, version 3.2* [or higher].

2 Similarities and differences with respect to SMURPH

The striking similarity between PicOS and SMURPH is in the thread model. In both environments, a thread describes a finite state machine with the state transition function specified in terms of event wait operations. The rules for aggregating such operations and waking up the threads based on the occurrence of the awaited events are practically identical. In SMURPH, viewed as a simulator, the awaited events are delivered by abstract objects called *Activity Interpreters*, while in PicOS they are triggered by actual physical phenomena (e.g, a packet reception, a character arrival from the UART, and so on).

The first significant difference between the two systems is in the interpretation of time flow. In SMURH, time is purely virtual, which means that formally nobody cares about the actual execution time of the simulation program, but only about the proper marking of the relevant events with virtual time tags. As in all event-driven simulators, the virtual time tags have nothing to do with real time. For example, a significant amount of calculations may be needed to advance the virtual time by a few microseconds, and no computation at all may be required to bypass several hours of idleness caused by no events in the system model. In the first case, the tiny advancement of the virtual time may take several hours of calculations, in the second case, the system may immediately jump several hours into the future in no time at all.

Consequently, the formal useful semantics of SMURPH and PicOS threads are different. The actual execution time of a SMURPH thread is essentially irrelevant (unless it renders the model execution too long to wait for the results) and all that matters is the virtual delays separating the artificially triggered events. For example, two threads in SMURPH may be semantically equivalent, even though one of them may exhibit a drastically shorter execution

[1] PicOS term for "applications."

time than the other (due to more careful programming and/or optimization). In PicOS, however, the threads are not (just) models but they run the actual thing. Consequently, the execution time of a thread may directly influence the perceived behavior of the PicOS node.

In this context, the following two assumptions make our endeavor worthwhile:

1. PicOS programs are reactive, i.e., they are practically never CPU bound. In other words, the primary reason why a PicOS thread is making no progress is that it is waiting for a peripheral event rather than the completion of a lengthy calculation.

2. If needed (from the viewpoint of model fidelity), an extensive period of CPU activities can be modeled in SMURPH by appropriately (and explicitly) delaying certain state transitions.[2]

Consequently, in most cases, we can ignore the fact that the execution of a PicOS program takes time at all and only focus on reflecting the accurate behavior of the external events. With this approximation, the job of porting a PicOS praxis to its VUE^2 model becomes reasonably simple. To further increase the practical value of such a model, SMURPH provides for the so-called *visualization mode* of its execution. In that mode, SMURPH tries to map the virtual time of modeled events to real time, such that the user has an illusion of talking to a real application. This is only possible if the network size and complexity allow the simulator to catch up with the model execution to real time. If not, a suitable slow motion factor can be employed. These issues are described in detail in the SMURPH manual.

While the syntax of PicOS threads is close to that of SMURPH processes, there are some differences. First of all, the language of SMURPH is C++, while PicOS is plain C. Second, a PicOS praxis is a one-node program, while the SMURPH model of that praxis must consist of multiple nodes running the same or possibly different programs. Additionally, the praxis code must be supplemented by the models of all those components of reality that are needed by the praxis to run. This brings about three issues:

1. Transforming the syntax of PicOS programs to that acceptable by SMURPH.

2. Putting multiple nodes, possibly with different PicOS praxes, under the umbrella of a single SMURPH program (model).

3. Modeling the peripheral equipment needed by the praxis; also interfacing and parameterizing such models.

The third part is provided as a library of models whose interiors need not be interesting to the developer. This is to say, the third issue does not overly affect the operation of rendering a PicOS praxis executable under VUE^2, as long as the necessary peripherals have their models in the library. This is because the interface (API) to those models looks **exactly** as the PicOS API to the real equipment. On the other hand, the first two issues come into play when the praxis is (re)organized for execution under VUE^2. Notably, with the right structure (explained in the next section), the praxis may exist in a single version, which is shared between PicOS and VUE^2. Most of the code is directly shared. There is no need to convert it from PicOS to VUE^2 or vice versa. **Thus, for all practical purposes, the emulated execution deals with the real code!**

[2] We are contemplating adding tools to VUE^2 that would make it possible to specify the timing of state execution time in a PicOS thread. Such a specification could indicate that the amount of CPU time needed to run through the state is not trivial. At this time, the prospective usefulness of such a feature is unclear.

3 The VUE^2 compiler

The program that turns a collection of PicOS praxes into an executable model of the network is in fact the *mks* program of SMURPH, i.e., the compiler of SMURPH models into executable programs. Here is the list of components needed to set up the complete platform. These components can be configured under Cygwin (Windows) or directly under a UNIX system, e.g., Linux.

1. The standard (vanilla) SMURPH/SIDE package constituting the emulation kernel of the system.

2. PicOS, which in addition to its current collection of VUE^2-compatible praxes provides many files that are directly used by VUE^2.

3. VUE^2-specific environment consisting of libraries and other files that in combination with PicOS sources and praxes contribute to VUE^2 models.

One special item included with the VUE^2 environment (point 3) is the somewhat inappropriately named *udaemon*. This is a Tk (*wish*) script implementing an interface (GUI) to VUE^2 models.

The exact way of setting up SMURPH and PicOS is described elsewhere. Here we shall focus on the VUE^2-specific aspects. For illustration, suppose that the three components have been unpacked at the same level (e.g., in the user's home directory). They fill three separate source trees rooted at directories VUEE, PICOS, and SIDE.

3.1 Notes on configuring SMURPH/SIDE with VUE^2

When setting up SMURPH/SIDE (according to the instructions in the manual), you have to specify, when requested by maker, the proper location of the include libraries needed by VUE2. This location is VUEE/PICOS. Thus, the relevant fragment of your conversation with maker when installing the package should resemble this:

```
Now, please enter the list of paths to 'include' libraries
(which can be absolute or relative to your home directory), each
path in a separate line. Enter an empty line when done. This
path:

        /home/pawel/SIDE/Examples/IncLib

is standard and need not be specified.  If you want to exclude
the standard path, enter '-' as the only character of an input
line.
VUEE/PICOS
```

where the last line (in italics) is your response (it assumes that VUE^2 has been unpacked in the home directory). Note that the default suggested by maker is retained, and one more (VUE^2-specific) include library is added. The standard (default library) is also needed as it provides the models of wireless channels used by VUE^2.

Defaults can be selected for the remaining installation parameters. The monitor, needed for the Java DSD applet is optional. The monitor connection of SMURPH is not required for interfacing VUE^2 models to external daemons, and its only practical advantage is for internal monitoring of models under intricate debugging.

It makes sense to use a non-default name for the *mks* program generated by *maker*, i.e., the actual compiler of VUE^2 models. The recommended name is *vuee* or *vue2* (anything starting with the three letters *vue* will do). When called under one of these names, the compiler automatically forces –L and –W, i.e., disables the (redundant) models of wired channels[3] and selects the visualization mode of SMURPH. Note that all VUE^2 models require –W to compile (while –L is optional).

It is possible to have a single installation of SIDE to be used for VUE^2 as well as for other purposes, i.e., simulation of network protocols. Having run *maker* once to create the *vuee* compiler, you can run it again possibly selecting another configuration of *include libraries* and assigning the standard (or different) name to the *mks* compiler. Depending on which compiler version is invoked, the corresponding *include libraries* will be referenced and the respective set of options will be applied.

3.2 Notes on setting up VUE^2

The VUE^2-specific components unpack into directory VUEE. It is convenient if this directory occurs at the same level as PICOS. Having unpacked the directory, move to VUEE/PICOS and execute *./mklinks* in that directory. That will set up a few links to PicOS files needed by VUE^2 and will only work if that directory occurs at the same level as VUEE. If this is unsuitable for whatever reason, you can edit the *mklinks* script specifying the correct path to PICOS.

Certain files in the PICOS tree (those linked to from VUEE/PICOS) are **directly** compiled as components of VUE^2 models. In particular, *tcv.c* (VNETI) is one of those files. Since release R090528A, the standard approach to augmenting VUE^2 with functionality carried over from PicOS is to transform the PicOS sources of those libraries, such that they are kept in one place (at PicOS). This greatly simplifies the compatibility issues for networked praxes.

Formally, the proper order of unpacking and installing the three systems is PICOS, VUEE, SIDE/SMURPH. This is because VUEE needs links to PICOS and SMURPH needs VUEE/PICOS as the include library. Having installed everything, you can compile those PICOS praxes that have been made VUE^2-compliant by moving to the praxis directory and executing *vuee* (or whatever name you assigned to the VUE^2/SMURPH compiler). The outcome of this compilation will be an executable file named *side* (*side.exe* under Cygwin). By running this file (with a suitable data file), you will execute the model.

A VUE^2 compliant praxis can be compiled by both the VUE^2 compiler as well as the PicOS compiler. In the latter case, you simply execute:

```
mkmk boardname
make
```

in the praxis directory. The PicOS compiler is able to accommodate multiple praxes in the same directory, even though this is seldom useful when viewed solely from PicOS's angle (therefore, this feature is not described in PicOS documentation). In the context of VUE^2, this is needed in situations when a single VUE^2 model must accommodate multiple praxes (different node types – see Section 4.6). Such praxes are usually strongly related (albeit different),[4] and then it may make sense to keep them in the same directory also for PicOS.

Traditionally, in the single-praxis-per-directory case, *mkmk* seeks a file named *app.c*, which it treats as the “anchor” file of the praxis. The configuration of its include files + the board options + the possible extra options specified in *options.sys* in the praxis directory determine the constituents of the target Image file to be loaded into the microcontroller. If there is no file named *app.c*, but there is another file whose name looks like *app_xxx.c*, *mkmk* assumes that this is the praxis root and compiles it instead. The resulting image files will be called

[3] Wireless and wired channel models can coexists, if needed. The system is capable of modeling hybrid networks.

[4] The generic Tags and Pegs model is a typical example, e.g., collectors and aggregators in EcoNet.

Image_xxx (the ELF version) and *Image_xxx.a43* (the Intel hex version). If multiple files named *app_...* are present, e.g., *app_one.c* and *app_two.c*, *mkmk* will assume that there are multiple praxes in the directory and generate a *Makefile* to compile them all into separate images (*Image_one* and *Image_two* in this case). A VUE^2-compliant configuration of multiple praxes will follow this convention to make those praxes discernible by *mkmk* (PicOS), while keeping them unified for the purpose of their combined model in VUE^2.

One more VUE^2-related feature of *mkmk* is the treatment of those source files whose names end with the suffix *.cc*. This suffix is needed for a file to be compilable by SMURPH. If a file with this suffix becomes needed by the praxis when compiled under PicOS, the compiler copies it to a similarly named file with the suffix *.c* and compiles it from there. This also applies to praxis anchor files, i.e.,files named *app_xxx.cc.*

4 Converting PicOS praxes to a VUE^2 compliant format

We start from general guidelines. On top of those guidelines are some further restrictions mostly resulting from the fact that not all PicOS operations are currently modeled in VUE^2. However, the ones that are modeled are sufficient for practically all our present praxes. Besides, the list of deficiencies will shrink as the modeled components are extended to accommodate new praxes (as the need arises). It is postulated that all serious new praxes be programmed in the VUE^2 compliant style from the very beginning.

4.1 General guidelines and comments

These guidelines can be viewed as prerequisites for an easy (mostly automatic) conversion of an existing praxis that was programmed with no VUE^2 compatibility in mind. This is to say, the first step in such a conversion would be to make sure that the praxis code follows these guidelines:

1. Construct your threads using these operations: *thread, strand, endthread, endstrand,* instead of *process* and *endprocess*. Similarly, use *runthread, runstrand* instead of *fork*.

2. Avoid *static* declarations within a process function unless you really need them. Such declarations are intended to express “permanent' local variables of processes. Instead, move them all to the global scope and use differentiating prefixes or suffixes in case of conflicts. Note that keeping all *de facto* global variables in one place helps the praxis avoid memory fragmentation on misalignment in its PicOS incarnation. Also note that, if truly needed, static declarations within processes can be accommodated (Section 4.5) with moderately complex extra tweaks.

3. Do not use *entry* statements with numerals or expressions, e.g.,

   ```
   entry (RS_CMND+1)
   entry (0)
   ```

   Make sure that all state arguments in those operations that accept state arguments are simple constants representing state names. This means that all states must be assigned distinct symbolic constants.

4. Identify those global (non-auto) variables that must be statically initialized. This includes those variables that are implicitly initialized to zero, if the praxis in fact assumes that they are initialized that way.

Many PicOS operations can be converted trivially to the corresponding SMURPH constructs using simple macros, e.g., *fork* to *create*, *entry* to *state*, and so on. Some operations require functions on SMURPH's side or macros that expand into multi-command sequences (e.g., *thread/strand* expanding into *perform*), but the conversion is still straightforward. The primary source of complications and the incurred tweaks related to points 2 and 4 is the fact that all global variables in the PicOS praxis must be turned into attributes of a node object in the VUE^2 model. This is because what looks like a complete program from the viewpoint of PicOS becomes merely a set of attributes and methods run by one node in the VUE^2 program.

Conceptually, the idea behind the PicOS-to-VUE^2 conversion is simple: PicOS threads are directly transformed into SMURPH processes run at stations representing network nodes. There are two types of variables referenced by such a process: local (automatic) variables (i.e., ones allocated on the stack and not surviving state transitions) and global (permanent) variables. The automatic variables can be left as they appear in the original PicOS code. The global variables must be turned into attributes of a SMURPH station representing the particular node. It makes no difference whether they are *static* in a thread or explicitly global. C++ cannot tell the difference between the two kinds. Moreover, as the code of a PicOS thread will (quite mechanically) become the code method of a SMURPH process, all *static* declarations must be removed from it, as their interpretation in the C++ context would be quite different from the one intended.

As mentioned above, it is possible to accommodate *static* variables of a PicOS process in VUE^2 in a way that will exactly agree with the original semantics of such a declaration. Namely, such variables can be turned into attributes of the corresponding SMURPH process, rather than into node attributes. This way they will survive state transitions while being invisible from outside of the process's code method. The only complication is some extra effort required to incorporate such variables into the two views of the praxis. This extra effort involves creating one or two more header files per each case of a process that insists on using static variables.

For illustration, we shall convert a simple praxis to VUE^2. In its original PicOS incarnation, the praxis occupies a single file, named *app.c* (not counting *options.sys*), and it has been programmed using minimalistic features of PicOS; in particular, all non-auto variables are global. The original version of the praxis, which can only be compiled for PicOS (using *mkmk* and *make*), can be found in directory PICOS/Apps/VUEE/ILLUSTRATION/SIMPLE. Here is the complete content of *app.c*:

```
#include "sysio.h"
#include "ser.h"
#include "serf.h"
#include "tcvphys.h"
#include "phys_cc1100.h"
#include "plug_null.h"

#define MAX_PACKET_LENGTH  60
#define IBUF_LENGTH        82

int           sfd = -1;
word          Count;
address       rpkt;

void show (word st, address pkt) {
       ser_outf (st, "RCV: %d [%s] pow = %d qua = %d\r\n",
              Count++,
              (char*)(pkt + 1),
              ((byte*)pkt) [tcv_left (pkt) - 1],
              ((byte*)pkt) [tcv_left (pkt) - 2]
       );
}

#define RC_TRY       0
```

```
#define RC_SHOW     1

thread (receiver)
  entry (RC_TRY)
       rpkt = tcv_rnp (RC_TRY, sfd);
  entry (RC_SHOW)
       show (RC_SHOW, rpkt);
       tcv_endp (rpkt);
       proceed (RC_TRY);
endthread

address spkt;

word plen (char *str) {
       word k;
       if ((k = strlen (str)) > MAX_PACKET_LENGTH - 5) {
              str [MAX_PACKET_LENGTH - 4] = '\0';
              k = MAX_PACKET_LENGTH - 5;
       }
       return (k + 6) & 0xfe;
}

#define SN_SEND     0

strand (sender, char)
  entry (SN_SEND)
       spkt = tcv_wnp (SN_SEND, sfd, plen (data));
       spkt [0] = 0;
       strcpy ((char*)(spkt + 1), data);
       tcv_endp (spkt);
       finish;
endstrand

char   *ibuf;

#define RS_INIT     0
#define RS_RCMD_M   1
#define RS_RCMD     2
#define RS_RCMD_E   3
#define RS_XMIT     4

thread (root)
  entry (RS_INIT)
       ibuf = (char*) umalloc (IBUF_LENGTH);
       phys_cc1100 (0, MAX_PACKET_LENGTH);
       tcv_plug (0, &plug_null);
       sfd = tcv_open (WNONE, 0, 0);
       tcv_control (sfd, PHYSOPT_TXON, NULL);
       tcv_control (sfd, PHYSOPT_RXON, NULL);
       if (sfd < 0) {
              diag ("Cannot open tcv interface");
              halt ();
       }
       runthread (receiver);
  entry (RS_RCMD_M)
       ser_out (RS_RCMD_M,
              "\r\nRF S-R example\r\n"
              "Command:\r\n"
              "s string  -> send the string in a packet\r\n"
       );
  entry (RS_RCMD)
       ser_in (RS_RCMD, ibuf, IBUF_LENGTH-1);
       if (ibuf [0] == 's')
```

```
              proceed (RS_XMIT);
  entry (RS_RCMD_E)
       ser_out (RS_RCMD_E, "Illegal command\r\n");
       proceed (RS_RCMD_M);
  entry (RS_XMIT)
       runstrand (sender, ibuf + 1);
       proceed (RS_RCMD);
endthread
```

Note that certain steps to ensure VUE^2-friendliness of our praxis have been taken already (operations *thread*, *strand*, and so on). In essence, a praxis like the one above, can be converted to VUE^2 mechanically. While presenting the necessary steps, we will be digressing into detailed explanations of the underlying issues and their solutions. Note that what truly matters is the "spirit" of the conversion, i.e., once you realize and understand what needs to be done, your favorite way of rendering a praxis VUE^2-compliant may differ in details. In particular, the exact names and structure of the files introduced in the process need not strictly follow our prescription.

4.2 SMURPH encapsulation

In PICOS/Apps/VUEE/ILLUSTRATION/SIMPLE_VUEE, you will find the converted version of our praxis (it can be compiled with *mkmk* as well as *vuee*). Note that all *.c* files have been assigned the *.cc* suffix to make them recognizable by SMURPH (and the C++ compiler) as parts of the model to compile.[5] The directory contains multiple files (five, not counting *options.sys* and the VUE^2 data file *data.xml*). Some of those files facilitate the structuring of the program into a VUE^2-compliant version (they are relevant for both views, i.e., PicOS and well as VUE^2). Some others are needed solely for SMURPH encapsulation, i.e., they are required to set up the environment for the VUE^2 model. The ones that fall into the latter category are:

node.h declares the SMURPH station type describing the node layout used by the praxis

root.cc encapsulates the VUE^2 library files that have to be compiled alongside the praxis code; also defines a single method needed by SMURPH's *Root* process to bring into existence (*create*) a node station

The layout of the two "encapsulation" files mentioned above is very simple and their contents are essentially fixed (i.e., praxis-independent); however, some of their fragments have the flavor of flexible parameters, which makes it difficult to turn those files into a truly fixed set of library modules. Here is our variant of *node.h*:

```
#ifndef __node_h__
#define __node_h__

#ifdef  __SMURPH__

#include "board.h"

station Node : PicOSNode {

#include "app_data.h"

       void appStart ();
```

[5] In a Cygwin installation of PICOS/SMURPH/VUEE, you should never put files ending with *.c* into a directory containing a VUE^2-ready praxis. The SMURPH/VUEE compiler **will remove** all such files at the end of its run! This is a nasty artifact of the fundamental problem with Cygwin under Windows, which ignores the case of letters in file names. The UNIX version of SMURPH compiler uses files ending with *.C* (capital C) as temporary files.

```
        void init () {

#include "app_data_init.h"

                appStart ();
        };
};

#include "stdattr.h"

#endif
#endif
```

This file should be included by all *.cc* files of the praxis, preferably as the very first header file. The first two lines make sure that the file will never be included more than once and, together with the closing *#endif*, comprise a standard envelope for header files. The role of the second *#if* is to ignore the entire contents of the file, if the including program is not being processed by SMURPH (VUE^2). Thus, the header will appear empty when compiled under PicOS. This is the standard way of telling the difference between the two environments.

The primary purpose of *node.h* is to declare the SMURPH station type representing a node in our virtual network. All such nodes are derived from the fundamental VUE^2 type *PicOSNode* providing the framework for any device running PicOS (type *PicOSNode* brings in the standard set of PicOS operations and attributes). File *board.h* is a library header containing all the necessary elements to make *PicOSNode* defined and accessible as a base type for *Node*.

Note that the node type need not be called *Node*. In particular, if your model accommodates multiple praxes, they all usually need different node types. This is because the whole concept of a node type (understood as a SMURPH station class) stems from the need to provide a frame (C++ class) for the global variables used by the praxis (and, usually, different praxes need different sets of global variables). In such a case, the directory of your model may contain multiple variants of *node.h*. (obviously with different names).

The actual declarations of the praxis's global variables are provided in the header file *app_data.h* (belonging to the praxis directory). The reason for putting them in a header file is to have them all in one place while being able to include them in a different manner in the PicOS incarnation of the praxis, where they should simply appear in one of the program files as straightforward C declarations. This is discussed in detail in the next section. Needless to say, you don't have to name that header file *app_data.h*. Generally, e.g., for a complex praxis whose code is split into several *.cc* files, there may be multiple logically separated chunks of variables stored in multiple header files.

The *Node* station declares two methods that play a role in the node's initialization. The simple responsibility of *appStart* is to create and run the *root* process of the praxis, which in PicOS is done automatically by the kernel after system startup. In a VUE^2 model, this involves the SMURPH *create* operation which needs access to the class type of the *root* process. In order to decouple the declaration of that SMURPH object from the declaration of the *Node* station (putting them into the same file would be overly restrictive), we allow the code of *appStart* to be provided in a different file. When you take a peek at *app.cc* (containing the essence of the praxis code, including its *root* process), you will this statement at the very end:

```
praxis_starter (Node);
```

which is a macro expanding into:

```
void Node::appStart () { create root; }
```

You see why *appStart* (despite its trivial content) cannot be implemented once for all as a library method (of *PicOSNode*). First of all, the exact definition of the *root* process cannot be known by the library. But even if it were, a model consisting of multiple praxes (and thus multiple node types) would have to start different (specific) variants of *root* for the different praxes. Consequently, *appStart* must be defined by the praxis, and it should be a method of the praxis-specific *Node* class.

The second method of *Node*, *init*, carries out the complete initialization of the node amounting to "hardware" reset. The only addition to the action already taken care of by *appStart* (which is invoked by *init*) is the initialization of global variables. The code performing this task is provided in one more header file, *app_data_init.h.* Typically, the contents of that file are strongly related to those of the first header file containing the declarations of the global variables. This is described in detail in the next section.

Finally, *node.h* includes *stdattr.h*, which is a library file containing macros that make the standard collection of attributes and methods of *PicOSNode* accessible as simple objects (syntactically equivalent to C functions and variables). By virtue of those macros, the essential code of the praxis (the instruction list of its program) need not be modified with respect to the PicOS version.

We should keep in mind that a node model undergoes a certain initialization stage that is more fundamental than hardware reset and whose counterpart for the real device would be its physical assembly. This preliminary stage of creating the node based on the input data (Section 5) is performed "internally" in a way of which the "proper" model of the praxis need not be aware. However, the complete model still has to mention this action explicitly. This is one of the two roles of *root.cc*, whose contents are listed below:

```
#include "node.h"
#include "board.cc"

process Root : BoardRoot {
        void buildNode (const char *tp, data_no_t *nddata) {
                create Node (nddata);
        };
};
```

The file specifies the body of method *buildNode* belonging to the standard *Root* process that every SMURPH model must define, which is responsible for building and starting the model. The method takes two arguments: a character string identifying the node type, and a pointer to a data structure providing the standard set of parameters for the node, which are (automatically) extracted from the data set (Section 5). You need not be aware of the layout of that structure. The primary reason why *buildNode* cannot be predefined as a library method is that more involved models (especially those consisting of multiple praxes – Section 4.6) may need to use slightly different code. Note that the first argument passed to *buildNode* is ignored in the above version. As all nodes in our model are the same type (this is a single-praxis model), there is no need to look at that argument at all.

The second role of *root.cc* is to include *board.cc*, which subsequently includes all the *.cc* components of the VUE^2 library that have to be compiled together with the program.[6]

[6] The reason why those components are not precompiled (and kept in a library) stems from SMURPH's paradigm of hard parametrization of its models (whose objective is to make those models as efficient as possible). Namely, depending on the exact configuration of parameters, the compiled versions of standard modules may (and usually do) differ. While this feature is considerably less relevant for VUE^2 than in the general case (VUE^2 models use a small subset of SMURPH parameters), it still gets in the way of building object libraries of VUE^2 modules.

4.3 Variables

At first sight, it would seem that the most natural way to transform a PicOS program into its VUE^2 model would be to turn all its functions into methods of *Node*. That can be accomplished by a relatively simple and mechanical transformation of the function headers. In consequence, the global variables (which have become *Node* attributes) would be accessible the same way as before – with no modification to the code.

Unfortunately, process code methods in SMURPH are not *Station* methods, but rather methods of the respective *Process* objects. This means that while a function (turned a *Node* method) could reference the node's “global” variables exactly as in PicOS, a process could not do that without playing some tricks. Once we accept that those tricks are unavoidable, they can also be played by the functions; thus, there is no absolute need to transform those functions into *Node* methods.

The trick consists in covering the names of global variables with *access macros* which expand into remote references to the corresponding *Node* attributes. In a SMURPH model, any piece of code executed as part of that model, be it a process state or a function called from that state, runs within the context of a specific *current* station (which means a node in VUE^2). That context is represented by a global pointer, named *TheStation*, which can be used as an explicit prefix qualifying a reference to any attribute of the current node. For example, if the type name of your node is *Node*, and *Count* is one of its attributes (i.e., “global” variables of the praxis), this string:

```
((Node*)TheStation)->Count
```

references that attribute in a way that can be consistently and unambiguously applied in all circumstances in which any fragment of the praxis model can run. This can be made transparent by defining a macro for every global variable referenced by the program, e.g.,

```
#define Count       (((Node*)TheStation)->Count)
```

To make it a bit easier, VUE^2 provides this shorthand:

```
#define _dan(a,b)   (((a*)TheStation)-> b)
```

which makes it sufficient to say:

```
#define Count       _dan (Node, Count)
```

You can easily introduce your own shorthand macro attached to your particular node type (e.g., putting this definition in *node.h*):

```
#define _d(b)       _dan (Node, b)
```

which will further reduce the amount of effort needed to provide the variable's access macro:

```
#define Count       _d (Count)
```

If all the *de facto* global variables appearing in the praxis have been made truly global (according to Section 4.1), the procedure of making them accessible to both views of the program goes as follows. First, you create a header file containing their pure declarations (file *app_data.h* in our example). You want to be able to use the same file as an insert into the declaration of your node object (see page 9), as well as to declare the variables for a PicOS program (as straightforward globals in a C file). The most problematic difference between the two cases is the initialization: while C allows such variables to be statically initialized, C++ requires that they be initialized dynamically (which is usually done in the object's constructor).

To avoid replicating the same declarations in multiple places (which would be annoying and error prone), we make the static initialization conditional. This is the *app_data.h* file from our simple praxis:

```
#ifndef __app_data_h__
#define __app_data_h__

int     sfd __sinit (-1);
word    Count;
address rpkt;
address spkt;
char    *ibuf;

#endif
```

The only part that needs an explanation is the declaration of variable *sfd*. Suppose that it must be initialized to –1. When the above file is interpreted by the C compiler (for PicOS), macro *__sinit* will expand into "= –1". Under VUE^2, the macro is void, i.e., it expands into nothing, thus ignoring the initialization.

Note that under C all the remaining variables from the above list will be implicitly initialized to 0 (or NULL). For those of them for which initialization matters (be it the default initialization to zero, or explicit initialization to something else), initialization code must be provided in VUE^2. This is the purpose of file *app_data_init.h* in our example praxis, which consists of these two lines:

```
sfd     = -1;
Count   = 0;
```

It contains the sequence of C++ statements needed to dynamically initialize those "global variables" (*Node* attributes) that actually have to be initialized. This file is only interpreted by VUE^2: the initialization for C is handled statically in the first header file.

One possible alternative approach to having multiple header files dealing with variables would be to have a single header file containing all the information about the global variables of the praxis and selecting its fragments needed for the different occasions (declaration, initialization) by defining symbols. This approach, which may be better suited for complex cases, is described in Section 4.6.

Note that generally the situation may be more complicated. For one thing, your program may be more complex than our simple illustration and you may need several header files describing different logically separable chunks of global variables (e.g., coming from different modules). Generally, each chunk of declarations requires a matching chunk of initializations (unless no variables in the chunk have to be initialized). Then, all those chunks have to be included in the proper places of the node object declaration (see page 9). Note that, by the virtue of that inclusion, they all share the same name space, which means that their names have to be strictly different, even if they come from different modules, where they could have been declared as (globally) *static*.

If this bothers you, remember that the mechanism for remapping references to global variables into ones to the corresponding node attributes described above makes it possible to mangle some names in the process. For example, consider this sequence:

```
#ifdef   __SMURPH__
#define mangle(a)   xx_ ## a
```

```
#else
#define mangle(a)   a
#endif
```

which you could insert somewhere into *node.h*, and then:

```
#ifndef __app_data_h__
#define __app_data_h__

int     mangle(sfd) __sinit (-1);
word    Count;
address rpkt;
address spkt;
char    *ibuf;

#endif
```

This will have the effect of transforming the name *sfd* into *xx_sfd*, but only in the VUE^2 version of the program. To complete the trick, you will have to modify the access macro for *sfd*, e.g.,

```
#define sfd   (((Node*)TheStation)-> xx_sfd)
```

for the respective module. Additionally, you can make such a declaration conditionally static, e.g.,

```
__STATIC int mangle(sfd) __sinit (-1);
```

The keyword in front expands into *static* for PicOS and into nothing for VUE^2. This way you can retain the (global) static qualifier from the C interpretation of the declaration, while ignoring it in VUE^2. Note that the *static* qualifier for a class attribute would have a completely different (and disastrous) meaning.

If you have a large number of modules, and you do not want to worry about guaranteeing the distinctiveness of their variable names, you can introduce per module mangler macros and use them in combination with *__STATIC*. VUE^2 uses such mangling schemes internally to avoid accidental conflicts among the names of internal attributes/methods of *PicOSNode* and those introduced by the praxis.

While one might be tempted to try to improve the way variable initialization is handled in the two views (e.g., by using the same code in both cases with the assistance of some convoluted macros), the fundamental problem is the difference in acceptable syntax. For example, consider this static declaration of a C array:

```
byte vals [] = {1, 9, 3, 8, 5, 5};
```

Transforming it into a VUE^2-equivalent, which involves putting this:

```
byte vals [6];
```

in one place, and this:

```
vals [0] = 1; vals [1] = 9; vals [2] = 3;
vals [3] = 8; vals [4] = vals [5] = 5;
```

in another, is significantly more than a simple mechanical tweak. Things would simplify considerably if we banned static initialization in PicOS (perhaps except the default one to zero, which could be handled automatically in VUE^2). That, however, would tend to inflate the praxis code in the real device environment, which we would very much like to avoid.

Note that there is a way in VUE^2 to initialize selected variables of selected nodes from the input data set (see Section 5.12). This models the physical action of burning some differentiating data into the code flash of a real node.

4.4 Putting it all together

The library of VUE^2 macros and functions added to SMURPH makes the adaptation of process code surprisingly easy. Let us have another look at the original code of our praxis being adapted for VUE^2 (page 7). Having removed the global variables from the code and having inserted:

```
#include “node.h”
```

in the front of the list of included headers, we can retain verbatim practically everything that remains. One little extra adjustment is the removal of the definitions of the symbolic constants representing process states. They have to be encapsulated into a conditional sequence contributing to the program's preamble. The complete new file can be found in PICOS/Apps/VUEE/ILLUSTRATION/SIMPLE_VUEE/app.c. Here is how it begins:

```
#include "node.h"
#include "sysio.h"
#include "ser.h"
#include "serf.h"
#include "tcvphys.h"
#include "phys_cc1100.h"
#include "plug_null.h"

#define  MAX_PACKET_LENGTH 60
#define  IBUF_LENGTH       82

#ifdef __SMURPH__

process receiver {
       states { RC_TRY, RC_SHOW };
       perform;
};

process sender {
       char *data;
       states { SN_SEND };
       perform;
       void setup (char *bf) { data = bf; };
};

process root {
       states { RS_INIT, RS_RCMD_M, RS_RCMD, RS_RCMD_E, RS_XMIT };
       perform;
};

#define      sfd    _dan (Node, sfd)
#define      Count  _dan (Node, Count)
#define      rpkt   _dan (Node, rpkt)
#define      spkt   _dan (Node, spkt)
#define      ibuf   _dan (Node, ibuf)

#else  /* PICOS */

#include "app_data.h"
```

```
#define       RC_TRY             0
#define       RC_SHOW            1

#define       SN_SEND            0

#define       RS_INIT            0
#define       RS_RCMD_M          1
#define       RS_RCMD            2
#define       RS_RCMD_E          3
#define       RS_XMIT            4

#endif /* VUEE or PICOS */
```

The interesting part, which we shall call the file's preamble, starts with *#ifdef __SMURPH__* and ends with the closing *#endif*. In the VUE^2 view, the program needs C++ object declarations for its processes. In the case of a PicOS *thread* (like *receiver* or *root*) converted for VUE^2 such a declaration is completely trivial and its information content amounts solely to providing the list of states (*perform* is a standard incantation of SMURPH to say that the process does have a code). For a *strand*, which maintains a pointer to its data structure, that pointer must be additionally declared as a process attribute (its type should agree with the second argument of the corresponding strand declaration – see page 8) and initialized in the process's setup method. The two cases listed above are definitive: this is all you have to do to turn any PicOS process into its VUE^2 model.

The second part of the VUE^2-specific section of the preamble is the list of access macros making the global variables (which have been turned into *Node* attributes) visible to the praxis (see Section 4.3). The PicOS section includes the file *app_data.h* (which directly declares the global variables together with their initializers) and defines the symbolic constants representing the states of PicOS processes.

The rest of the code looks exactly as in the original praxis. One line added at the very end is the reference to *praxis_starter* whose role was discussed on page 10).

4.5 Static variables of processes

One advantage of having a static variable declared within a process, as opposed to turning it into a global variable, is that it exists in a separate name space. For example, multiple processes can use the same name for their private instances of that variable (e.g., *packet* representing a packet pointer used by the sender and the receiver).

From some point of view, handling such variables is simpler than handling general global variables of the praxis because there is no need to provide access macros for them (like the ones discussed in Section 4.3). A static variable of a process can only be referenced from the process's code method, which means that declaring it as a process attribute is going to do the whole trick. Thus, the only extra complexity is the need to create two additional header files per each process using static variables: one to contain their declarations, the other to provide the initialization code for those of them that have to be initialized. The rules are the same as for global variables turned node attributes, except that the keyword *__STATIC* (page 14) must now appear in front of every declaration. For illustration, consider this possible variant of the receiver process from our sample praxis:

```
thread (receiver)

  static address packet;

  entry (RC_TRY)
       packet = tcv_rnp (RC_TRY, sfd);
  entry (RC_SHOW)
```

```
       show (RC_SHOW, packet);
       tcv_endp (packet);
       proceed (RC_TRY);

endthread
```

For the VUE^2 view of the process, the declaration of *packet* should be moved to the SMURPH process object, where the variable should appear as an attribute with the keyword *static* stripped off. You will thus create a header file, e.g., named *receiver_data.h*, and make it look like this:

```
#ifndef __receiver_data_h__
#define __receiver_data_h__

__STATIC address packet;

#endif
```

Then, you will include that file in the preamble, inside the SMURPH declaration of the process:

```
#ifdef __SMURPH__

process receiver {

#include “receiver_data.h”

       states { RC_TRY, RC_SHOW };
       perform;
};
...
```

You should also include the same file for the PicOS view of the process replacing the original declaration of the variable, i.e.,

```
thread (receiver)

#include “receiver_data.h”

  entry (RC_TRY)
       packet = tcv_rnp (RC_TRY, sfd);
  entry (RC_SHOW)
       show (RC_SHOW, packet);
       tcv_endp (packet);
       proceed (RC_TRY);
endthread
```

It would seem that the latter inclusion should be conditional – to make sure that the file will not be included in that place when the program is processed by VUE^2. Note, however, that the first inclusion of *receiver_data.h* precludes the second inclusion (the header file is guarded against multiple inclusions). Thus it all resolves itself automatically: under PicOS, the first inclusion does not happen, so the second one is active; then, under VUE^2, the first inclusion is active and the second one is void, exactly as needed.

In the above case, the static variable need not be initialized, so there is no need for the second (initializer) header file. Otherwise, you would do it in exactly the same way as for a global variable that has become a node attribute. The initializer header would be included in the process's setup method (which you would have to provide in that case), e.g.,

```
#ifdef __SMURPH__
```

```
process receiver {

#include “receiver_data.h”

       states { RC_TRY, RC_SHOW };
       perform;

       void setup () {

#include “receiver_data_init.h”

       };
};
...
```

Note that if several processes share the same configuration of static variables (which is usually the primary reason why you want to keep them this way), you can create a single pair of headers: definition/initializer for all of them. But then, you will have to be able to include them multiple times, possibly in the same program file, in a selective sort of way, which renders the operation mildly cumbersome. One way out is to go like this:

```
#ifdef __SMURPH__

process receiver {
#include “xcv_data.h”
       states { RC_TRY, RC_SHOW };
       perform;
};

process sender {
#include “xcv_data.h”
       char *data;
       states { SN_SEND };
       perform;
       void setup (char *bf) { data = bf; };
};
#define __xcv_data_h_no_more__
...
```

with *xcv_data.h* containing this:

```
#ifndef __xcv_data_h_no_more__

__STATIC address packet;

#endif
```

The problem doesn't concern the initializer header (if one is required), because such files are not protected against multiple inclusion and they are only included in the VUE^2 section of the preamble.

4.6 Models of multiple praxes and other complex cases

Subdirectories SIMPLE_SND and SIMPLE_RCV of PICOS/Apps/VUEE/ILLUSTRATION contain two independent (and different) praxes derived from the original sample praxis (in directory SIMPLE), such that one of them can only act as a sender while the other one can only receive. In other words, each of the two praxes only runs one of the two processes in the original praxis. We shall combine them into a single model (directory SIMPLE_TWO). The praxes in that directory can be compiled with PicOS as well as with VUE^2.

The general idea is to keep two independent sets of files in a way that separates the two praxes completely, except for a kludge needed to turn them into a single SMURPH (VUE^2) program. The names of files belonging to one praxis include the tag _*snd*, and those belonging to the other praxis contain _*rcv*. In addition to *options.sys* and the XML data file (which are shared by the two praxes), file *root.cc* is the single component of the actual program that links the two sets of files together.

One issue surfacing in this context is the need to separate the names of processes used by the two praxes, especially that there is at least one process name that they absolutely have to share, i.e., the *root* process required by PicOS. This is formally impossible in VUE^2: the process name translates into a class name, and an attempt to link a model containing two different process code methods of the same class will clearly fail. Of course, it would be nice if the praxes could generally separate the name spaces of their processes, as it is not uncommon to use some typical and standard process names in different praxes (like *reader*, *sender*, *receiver*, *monitor*, and so on).

Another observation is that the simple approach described in Section 4.4, whereby the SMURPH declarations of a process appears solely in the file that defines its code method, is not extremely flexible. It is not going to work if the code method must be referenced in another file. The problem is in fact more general and concerns a consistent way of referencing objects (processes, variables, functions) across multiple files that would work for both views of the praxis (or multiple praxes kept in the same directory).

The combination of two praxes in directory SIMPLE_TWO illustrates one attempt at accomplishing the objectives mentioned above with a reasonable mixture of simplicity, readability, and generality. The equivalents of file *node.h* from the single praxis case have been replaced with files *vuee_snd.h* and *vuee_rcv.h*. Each of those header files declares the node framework for VUE^2 and resolves the global naming issues of the respective praxis with respect to its two views (SMURPH/PicOS) as well as with respect to the other praxes coexisting within the model. Here is one of those headers:

```
#ifndef __vuee_snd_h__
#define __vuee_snd_h__

#ifdef  __SMURPH__

#define  THREADNAME(a)   a ## _snd

#include "board.h"

station NodeSnd : PicOSNode {

#define  __dcx_def__
#include "app_snd_data.h"
#undef   __dcx_def__

       void appStart ();
       void reset () { PicOSNode::reset (); };
       void init ();
};

#define  _daprx(a)       _dan (NodeSnd, a)

#include "stdattr.h"

strandhdr (sender, NodeSnd) {
       char *data;
       states { SN_SEND };
       perform;
```

```
        void setup (char *bf) { data = bf; };
};

threadhdr (root, NodeSnd) {
        states { RS_INIT, RS_RCMD_M, RS_RCMD, RS_RCMD_E, RS_XMIT };
        perform;
};

#else  /* PICOS */

#include "sysio.h"

#define      SN_SEND      0

#define      RS_INIT      0
#define      RS_RCMD_M    1
#define      RS_RCMD      2
#define      RS_RCMD_E    3
#define      RS_XMIT      4

procname (sender);
procname (root);

#endif /* SMURPH or PICOS */

#define  __dcx_dcl__
#include "app_snd_data.h"
#undef   __dcx_dcl__

#endif
```

Note that it consists of two separate parts: one for the VUE^2 view, the other for PicOS. For one thing, the declarations of processes have been made global to facilitate cross referencing from all files that include the header (for PicOS, this is handled by the *procname* announcers). Additionally, the name spaces of the processes (in the VUE^2 view) have been separated. That has been accomplished by the *THREADNAME* macro, which assigns a suffix (*_snd* in this case) to all process names defined and referenced by the praxis in its VUE^2 variant. To make this work, the SMURPH declarations of processes use the macros *strandhdr* and *threadhdr*, (instead of the SMURPH keyword *process*), which invoke *THREADNAME* to apply the differentiating suffix to the names of the declared processes. Note that this name separation is only needed for VUE^2: under PicOS, the praxes will be compiled as two completely separate programs.

Instead of having two different files (or, generally, two sets of files) pertaining to the praxis global data (declaration, initialization – Section 4.3), we opt for a single-file with several sections selectable by defining constants. This is the header *app_snd_data.h* being included in two places by *vuee_snd.h*. The constants wrapping up those inclusions select the kind of information we need at the particular point. We call this information the *data context*. These are the contents of *app_snd_data.h*:

```
// Definition context
#ifdef  __dcx_def__

#ifndef __app_snd_data_defined__
#define __app_snd_data_defined__

int     sfd __sinit (-1);
address spkt;
char    *ibuf;

#endif
```

```
#endif

// Declaration context

#ifdef __dcx_dcl__

#ifdef __SMURPH__

#define      sfd    _daprx (sfd)
#define      spkt   _daprx (spkt)
#define      ibuf   _daprx (ibuf)

#else

extern int sfd;
extern address spkt;
extern char *ibuf;

#endif

#endif

// Initialization context

#ifdef __dcx_ini__

        sfd = -1;

#endif
```

Basically, there are three contexts. In addition to the two contexts described earlier (Section 4.3), i.e., definition and initialization, we explicitly identify yet another context called declaration. For VUE^2, that context brings in the macros covering (and possibly mangling) variable names, such that their references in the praxis code are properly turned into references to the respective node attributes. For PicOS, the declaration context makes the variables external in order to be reachable across different files of the praxis. The logical effect is the same in both cases: the variables become accessible, even though their actual definition may occur in a different (single) place. Note that the definition context is guarded by an additional constant to make sure that the same component of the praxis can only define the variables once, even though the header file may be included several times.

When you execute *mkmk* in SIMPLE_TWO followed by *make*, you will obtain two PicOS Image files: *Image_snd* and *Image_rcv* (see Section 3.2). For VUE^2, the two (generally multiple) sets of files must be joined at some point into a single program. This is the responsibility of the shared *root.cc* file:

```
#include "sysio.h"
#include "board.h"
#include "board.cc"

void buildSndNode (data_no_t*);
void buildRcvNode (data_no_t*);

process Root : BoardRoot {

        void buildNode (const char *tp, data_no_t *nddata) {
                if (strcmp (tp, "snd") == 0)
                        buildSndNode (nddata);
                else
                        buildRcvNode (nddata);
        };
};
```

Compare its contents to those from the single-praxis case (page 11). One difference is that the node type (*tp*) argument of *buildNode* is now used: different functions are invoked to create nodes belonging to the different praxes. Moreover, in order to make the two praxes separable, the file never references the actual node types. Instead, that task is delegated to two global functions whose headers can be provided without knowing the node types. For the same reason, the present version of *root.cc* includes neither *vuee_snd.h* nor *vuee_rcv_h*.

The global functions responsible for creating nodes for the two praxes are defined in *vuee_snd.cc* and *vuee_rcv.cc*, which also contain the *init* methods for the nodes. *Here is vuee_snd.c:*

```
#include "vuee_snd.h"

void NodeSnd::init () {

#define  __dcx_ini__
#include "app_snd_data.h"
#undef   __dcx_ini__

       // Start application root
       appStart ();
}

void buildSndNode (data_no_t *nddata) {

       create NodeSnd (nddata);
}
```

The node type identifiers used by *buildNode* (in *root.cc*) to decide which type of node is being created come from the input data set (Section 5.3).

4.7 Praxis options

File *options.sys* used by PicOS to select the optional components of the PicOS kernel and some of its parameters is also included and interpreted by VUE^2. Most of the PicOS options, however, have no meaning in VUE^2. For example, items like selecting the CPU clock are blatantly useless in a VUE^2 model. Also note that options selecting the UART rate, the LBT parameters for the transceiver, and so on, are settable in the input data set (on the per-node basis), so VUE^2 does not look for them in *options.sys*.

One PicOS option that retains a part of its meaning in *options.sys* is radio module selection. It is not needed as much by VUE^2 as by the networking components of PicOS (*net.c*) that compile with VUE in their original versions. At present, you can only select CC1100 and DM2200 (because *phys_cc1100* and *phys_dm2200* are the only RF modules present in the VUE^2 library). This is very easy to extend, should a need arise.

Note that there is no way to tell VUE^2 that its should make the node fit a particular board (as described in the BOARDS directories). Again, this kind of specification for VUE^2 belongs in the input data set.

One important use of *options.sys* for VUE^2 is to describe the coarse-grained configuration of components from the VUE^2 library to be compiled into the node class. These components are specified via constants whose values are irrelevant (what matters is the definition or its absence). When inserted into *options.sys* such definitions can be encapsulated into:

```
#ifdef __SMURPH__
```

```
...
#endif
```

but they don't have to, as those constants are not interpreted under PicOS. Here is the complete list of options:

VUEE_LIB_PLUG_NULL	compiles in the NULL plug for VNETI
VUEE_LIB_PLUG_TARP	compiles in TARP
VUEE_LIB_XRS	compiles in the library for External Reliable Scheme over UART
VUEE_LIB_OEP	compiles in the Object Exchange Protocol library
VUEE_LIB_LCDG	compiles the Nokia LCD display support (N6100P)

For example, this sequence in *options.sys*:

```
#define       VUEE_LIB_PLUG_NULL
#define       VUEE_LIB_PLUG_XRS
```

makes sure that support for the NULL plugin and XRS is compiled in.

Note that the compiled-in structure of the base node type (class PicOSNode) is the same for all nodes, even if the model consists of multiple praxes. This is generally not a problem because the praxis code need not use those components that it doesn't care about.

5 Input data

The input data file parameterizing a VUE^2 model follows an XML format. Below is a complete sample data file describing a three-node network:

```
<network nodes="3" radio=”3” port=”2234”>
    <grid>0.1m</grid>
    <channel>
        <shadowing bn="-110.0dBm" syncbits="8">
             RP(d) = received power at distance d
             XP    = transmitted power
             X     = lognormal random Gaussian component
             =======================================================
             RP(d)/XP [dB] = -10 x 3.0 x log(d/1.0m) + X(1.0) - 38.0
             =======================================================
       </shadowing>
       <cutoff>-120.0dBm</cutoff>
       <ber>
             Interpolated ber table:
             ======================
               SIR          BER
             50.0dB       1.0E-6
             40.0dB       2.0E-6
             30.0dB       5.0E-6
             20.0dB       1.0E-5
             10.0dB       1.0E-4
              5.0dB       1.0E-3
              2.0dB       1.0E-1
              0.0dB       2.0E-1
             -2.0dB       5.0E-1
             -5.0dB       9.9E-1
       </ber>
       <frame>12 0</frame>
       <rates boost=”yes”>
```

```
         0      9600           6.0dB
         1      38400          0.0dB
         2      200000       -10.0dB
   </rates>
   <power>
         0      -30.0dBm
         1      -15.0dBm
         2      -10.0dBm
         3       -5.0dBm
         4        0.0dBm
         5        5.0dBm
         6        7.0dBm
         7       10.0dBm
   </power>
   <channels number=”255”>
         separation
         20dB 29dB 38dB 46dB 54dB 62dB 70dB 78dB 86dB 94dB 102dB
   </channels>
   <rssi>
         0      -202.0
         255      53.0
   </rssi>
</channel>
<nodes>
   <defaults>
       <memory>1124 bytes</memory>
       <processes>16</processes>
       <radio>
         <power>7</power>
         <rate>1</rate>
         <boost>1.0dB</boost>
         <channel>4</channel>
         <preamble>32 bits</preamble>
         <lbt>
                delay               8msec
                threshold           -109.0dBm
         </lbt>
         <backoff>
                min                 8msec
                max                 303msec
         </backoff>
       </radio>
       <uart rate="9600" bsize="12" mode=”direct”>
         <input source="socket"></input>
         <output target="socket" type="held"></output>
       </uart>
       <eeprom size="524288" clean="00">
         <chunk address=”0”>00 01 02 04 08</chunk>
         <chunk address=”8192” file=”eblock.bin”></chunk>
       </eeprom>
   </defaults>
   <node number="0">
       <location>1.0 4.0</location>
       <preinit tag="ESN" type="lword">0x8000ff01</preinit>
   </node>
   <node number="1" start=”off”>
       <location>1.0 10.0</location>
   </node>
   <node>
       <eeprom size=”10485760,20480” image=”node2_eeprom.img”
          clean=”00” erase=”page” overwrite=”no”
          timing="0.000004,0.000006,
                  0.000004,0.000034,
                      0.000030,0.000038,
                      0.000005,0.040000">
           </eeprom>
           <iflash size=″512,2″ clean=″ff″></ifash>
           <sensors>
              <input source=″socket″></input>
              <sensor vsize=″2″ delay=″0.05″>4095</sensor>
              <sensor vsize=″2″ delay=″0.1,0.5″ init=″3″>999</sensor>
              <actuator vsize=″4″ delay=″0.01″ init=″0″></actuator>
           </sensors>
           <location>1.0 10.0</location>
           <ptracker>
               <output target="socket"/>
               <module id="cpu">0.3 0.0077</module>
               <module id="radio">0.0004 16.0 30.7 30.7</module>
               <module id="storage">
                      0.030 0.030 10.0 15.0 17.0 16.0
               </module>
               <module id="sensors">0.0 2.5</module>
           </ptracker>
        </node>
    </nodes>
    <roamer>
       <input source=″string″>
             R 2 [1.0 1.0 21.0 24.0] [0.5 1.8] [1.0 6.0] -1
       </input>
    </roamer>
    <panel>
       <input source="string">T 10.0\nO 1\nT +5.0\nF 1</input>
    </panel>
</network>
```

Illustration 1: A sample data file.

Each data file describes a single <network>. The first attribute of the <network> element (*nodes*) is mandatory and specifies the total number of nodes (of all types). The second attribute (*radio*), if present, specifies explicitly the number of nodes that will be interfaced to the RF channel. If this attribute is missing, it defaults to the total number of nodes, unless the data set contains no specification of a radio channel, in which case it defaults to zero. These numbers remain fixed during the execution, but the nodes may exhibit highly dynamic behavior including mobility.

One more (optional) attribute of <network> is *port*, which can be used to assign a non-standard port number to the socket opened by the program for connections from agents (see Section 8).

Some input elements include numbers. Typically, besides numbers, such an element may contain non-numeric text, which is ignored (treated as a comment). For example, the only relevant items from the <shadowing> element in <channel> are the numbers -10, 3.0 ,1.0, 1.0, 38.0. Also, any non-numerical characters in the otherwise numerical arguments of elements (like the letters dBm in "-110.0dBm") are ignored. Thus, the equivalent comment-free specification of <shadowing> is:

```
<shadowing bn="-110.0" syncbits="8">-10 3.0 1.0 1.0 38.0</shadowing>
```

Note that the minus sign apparently preceding 38.0 has been ignored: this is because of the space separating it from the first digit.

In the subsequent sections, we shall discuss the sub-elements of <network>.

5.1 Grid

This optional element has no arguments, and its text must include a single (floating point) number. This number specifies the granularity (in meters) of the coordinate grid for node deployment and movement. By default, the value of grid is 1.0, which means that node location will be rounded to full meters. This parameter directly determines the discrete granularity of virtual time in SMURPH (the ITU), which is equal to the amount of time required for a radio signal (propagating at 299,792,458 m/s) to cross the grid unit.

5.2 Channel

This element describes a radio channel. It used to be mandatory, but, as of release R090526A, is optional, which means that you can also model in VUE^2 nodes that are not equipped with a radio interface. If no radio channel is present in the data set, then no nodes are interfaced to the radio, which also means that a radio attribute of <network> specifying a nonzero number of nodes is illegal.

It is anticipated that the population and format of sub-elements of <channel> will expand as new types of channel models are added to the library. The element itself takes no arguments, and the exact characteristics of the channel are described by the sub-elements. The only channel model available at present is *shadowing*, whose propagation properties are described by two numerical arguments of the <shadowing> element and five numbers that must occur in its body. Argument *bn* specifies the assumed level of background noise in dBm, and *syncbits* is the minimum number of correctly received preamble bits that a receiver must perceive in front of a recognizable packet. Both arguments are required (they have no defaults).

Signal attenuation in the shadowing channel model is described by the following formula:

$$\left[\frac{P_r(d)}{P_r(d_0)}\right]_{dB} = -10\beta\log\left(\frac{d}{d_0}\right) + X(\sigma_{dB})$$

where $P_r(x)$ is the received signal level at distance x , d_0 is a (intentionally small) reference distance, β is the loss exponent and X is a Gaussian random variable with zero mean and standard deviation of σ_{dB} . For our purpose, we transform the formula to give us signal attenuation at distance d, which we can directly plug into the respective assessment method in SMURPH:

$$\left[\frac{P_r(d)}{P_x}\right]_{dB} = -10\beta\log\left(\frac{d}{d_0}\right) + X(\sigma_{dB}) - L(d_0)$$

where P_x is the transmission power and $L(d_0)$ is the calibrated loss at the reference distance d_0 . This formula is is only meaningful when $d > d_0$. It is tacitly assumed that transmission at a shorter distance incurs the same attenuation as a transmission at the reference distance d_0 . This is exactly the formula from the <shadowing> element on page 23, i.e.,

```
    RP(d)/XP [dB] = -10 x 3.0 x log(d/1.0m) + X(1.0) - 38.0
```

with the parameter values: $\beta = 3.0$, $d_0 = 1.0\text{m}$, $\sigma_{dB} = 1.0$ and $L(1\text{m}) = 38\text{dB}$. This formula is used do determine the signal level at a receiving node N_r .

The <cutoff> element specifies a single floating point number interpreted as the so-called cut-off signal level in dBm. This is a threshold for the received signal level below which it can be safely assumed that there is no signal at all. This value is used to calculate the cut-off distance restricting the population of neighborhoods in the model, which may impact the execution speed. This element is mandatory.

All signals arriving from multiple transmitters within the cut-off range of N_r combine additively. The SIR of a single signal perceived by N_r is equal to the value of that signal level at N_r divided by the sum of all the other signals arriving at N_r augmented by the ubiquitous background noise (the *bn* argument of <shadowing>). Then, the table from the <ber> element is consulted to determine the probability of a bit error. That table is an array of numbers occurring in pairs. The numbers in the first column must be decreasing, while those in the second column must be non-decreasing (normally they are increasing). This is a discrete specification of a function that translates the signal to interference ratio (SIR) at the receiver into a bit error rate (BER), i.e., the probability that a single bit is received in error. The smooth BER function for all possible values of SIR is obtained by interpolating the table. If the SIR is greater than or equal to the first value in the table, then the bit error rate is determined by the first entry (for example, it never decreases below 10^{-6} in the channel described in Illustration 1). If it less than the last value, then the bit error rate is 1, i.e., reception is impossible.

The bit error rate applies to the so-called “physical bits,” which are also used in determining the effective transmission rate. The three numbers within the <frame> element stand for the *bit* rate, *bits* per byte, and the number of extra *bits* needed to frame a complete packet. The rate parameter determines the number of physical bits transmitted per second. The second parameter of that element translates physical bits into logical bits by indicating the number of physical bits in one byte (octet). This mapping accounts for the encoding, e.g., 6-bit symbols encoding 4-bit nibbles. The last parameter (also expressed in physical bits) covers any special (non-preamble) components of the frame that are invisible to the receiver software but contribute to the overall length of a transmitted and received packet, e.g., a start symbol.

Refer to the SMURPH manual for an explanation of the dynamics of the interference model. The assessment method responsible for detecting the beginning of a packet at a receiver (*RFC_bot*) checks if at least *syncbits* (physical) bits of the preamble (see page 23) immediately preceding the first bit of the actual packet have been received without an error, according to the bit error rate calculated as explained above. The second assessment method (*RFC_eot*), determining the final success of a packet reception, is not used in the channel model. Instead, the receiver invokes *RFC_erd* to await the first bit-error event in the packet. The (reasonable) simplification assumed in the model is that the first error bit will render its symbol invalid, which will interrupt the reception. This is warranted by the fact that all the legitimate symbols used by DM2200 (and most other RF modules) have the property that flipping a single bit renders the symbol invalid.

Clearly, the <channel> element describes a bit more than just the radio channel. Some of the parameters there are related to the characteristic of the RF modules used by the nodes. Although, arguably, it might make sense to associate those attributes with individual transceivers (and this may happen in the future), the present description favors networks based on the same (or similar) equipment used by all nodes (at least within the realm of a single “channel”). It is possible to have multiple channels possibly interfaced by nodes acting as gateways.

Owing to the fact that all RF drivers in PicOS share practically the same logical structure, it makes sense to encapsulate their hardware-specific properties into parameters that can be set in the input data file. This way, functions like *phys_cc1100, phys_dm2200* point to essentially the same driver model, while the proper setting of the channel parameters in the input data should bring that model close to the intended hardware.

These sub-elements of <channel>: <power>, <rates>, and <channels> describe the set of options for power setting, bit rates, and channel numbers available to the nodes. In particular, the first column of numbers in the <power> element lists the discrete values usable by the praxis as arguments of *PHYSOPT_SETPOWER* with the corresponding settings of the actual transmission power for the RF module (in dBm) appearing in the second column. In a similar way, <rates> declares the selection of bit rates available to *PHYSOPT_SETRATE*. If the *boost* attribute of the <rates> element is *yes* (as in Illustration 1), then the third column of numbers specifies rate-specific factors (boosts) applied to the signal at reception. This way, different rates may result in different effective transmission ranges. If the *boost* attribute is absent (or different from *yes*), the specification consists of two columns and there are no rate specific boost factors (equivalent to 0dB boost for all rates).

If a <channels> element is present, it describes the number of channels settable by the praxis with *PHYSOPT_SETCHANNEL*. The optional sequence of FP numbers appearing within the text of that element refers to channel separation in dB. Normally, these numbers are increasing: the first number gives the separation between two adjacent channels, the second between channels n and n+2, and so on. The separation becomes infinite when the numbers run out. In particular, if no numbers are specified at all, the separation between any pair of different channels is infinite.

Elements <power> and <rates> are mandatory. In a simple case, e.g., if there is only one available power setting, the <power> element may contain a single FP number: the power setting with the implied index of zero. On the other hand, <channels> is optional. If absent, there is only one channel shared by all nodes, and its number is zero.

The optional <rssi> element describes the way of transforming the received signal strength into RSSI indications (appended at the end of a received packet). If no <rssi> element occurs within <channel>, no RSSI indications will be returned to the praxis (the corresponding packet byte will be always zero). The first column of numbers in the <rssi> element refers to the indications returned to the praxis, while the numbers in the second column describe the actual received signal levels in dBm. Values in between will be interpolated.

5.3 Nodes

This element describes the configuration of nodes in the network. Each of its sub-elements, except <default>, provides a set of parameters for one specific node. The expected contents of <defaults> are the same as for <node> and describe the default setting of parameters for all nodes. If a given parameter is not explicitly mentioned within a <node> element, its <default> setting is assumed for the respective node.

A <node> accepts three optional attributes. One of them (the keyword *number*) specifies the node number. The node numbers are internal SMURPH identifiers of the *stations* implementing the nodes. Although nodes can be specified in any order in the data file, the resultant numbering of nodes must be continuous and start from zero. The total number of node definitions (<node> elements) in a data set must be equal to the *nodes* attribute of <network> (page 23).

An explicit number attribute of a <node> element, if present, makes it obvious how to match the definition to an actual node number in the network model. If the number attribute is absent, it is assumed that the definition has an implicit number tag equal to the number of the previous definition + 1. If the first <node> definition has no number attribute, it is assumed to refer to node 0. Note that it is legal for the input data set to contain noncontiguous chunks of node definitions. In any case, to be correct, they should exhaust all node numbers (between 0 and *nodes*-1) and do not attempt to describe the same node more than once.

Another attribute of <node> (keyword *type*) is a piece of string identyfying the node type. The value of this attribute is passed as the first argument to buildNode (Section 4.6) and is useful in models with multiple praxes.

One more (optional) attribute of <node>, which is also applicable to <defaults>, is *start*, which can be "on" or "off", with "on" being the default. If the value of start is "off" for a node, then the node will not be started automatically (*init* will not be called for it after *setup* – see page 9). Such a node will have to be started explicitly, either from a panel process (described by a <panel> element in the input data – see Section 7) or by an external agent.

Here is the list of sub-elements of <node> (and also <defaults>). The nontrivial elements are discussed later in separate sections.

```
<memory> ... memory size in bytes ... </memory>
```

This element declares the amount of memory (standard RAM) available at the node for *malloc*. This is equal to the physical size of RAM at the microcontroller minus the combined size of global variables (with provision for alignment). PicOS reports this amount upon startup as the so-called leftover RAM.

```
<processes> ... process number limit ... </processes>
```

This element declares the size of the process table, i.e., sets the limit on the maximum number of PicOS processes that may be present at the same time. If the element is absent, there is no explicit limit on the number of processes.

If a node is equipped with radio interface, then its declaration should include a <radio> element, which, in turn, should mention the following attributes:

```
<power> ... power index ... </power>
```

This element declares the initial setting of the transmission power with reference to the <power> table specified for the channel. The value must be one of the indexes occurring in the first column of the channel's <power> table. If the element is missing, the lowest index from the <power> table is implicitly assumed.

```
<rate> ... rate index ... </rate>
```

This element declares the initial setting of the transmission rate according to the <rate> table for the channel. The value must be one of the indexes occurring in the first column of the channel's <rate> table. If the element is missing, the lowest index from the <rate> table is implicitly assumed.

```
<channel> ... channel number ... </channel>
```

This element declares the initial channel setting for the RF module. The value must be a valid channel number according to the <channels> specification for the channel. If the element is missing, channel 0 is assumed by default.

```
<boost> ... receiver gain ... </boost>
```

This element declares the receiver gain (boost) in dB, which is 0.0dB by default.

```
<preamble> ... preamble length in physical bits ... </preamble>
```

This element specifies the packet preamble length in physical bits to be inserted by the node's transmitter in front of every packet.

```
<lbt> ... delay ... threshold ... </lbt>
<backoff> ... minimum ... maximum ... </backoff>
```

These parameters (four numbers) are used by the collision avoidance procedure of the node's transmitter. For <lbt> (Listen Before Transmit), the first (integer) number gives the time interval (in milliseconds) during which the transmitter will pause before a spontaneous transmission while monitoring the signal level. If the level is above the threshold (in dBm), the transmitter will back off randomly and try again. The back-off delay is between the minimum and maximum specified in the <backoff> element (both numbers are in milliseconds and must be integers).

The total number of nodes whose specification includes a <radio> element must match the number of nodes declared with the <network> element as being equipped with a radio interface. Note that this is usually the total number of nodes in the network, unless the <network> element has a non-trivial *radio* attribute. if the <default> element specifies a radio interface (has a <radio> sub-element), then any node devoid of <radio> is automatically assigned the default specification. If you want to explicitly indicate that a particular node is NOT equipped with a radio, use an empty <radio> element, e.g.,

```
<radio></radio>
```

This element is mandatory for a node equipped with radio:

```
<location> ... x ... y ... </location>
```

and it must be explicitly present in every such a <node>. There can be no default for a node's location: the <location> element is ignored if occurring in <default>. The element assigns an initial location to the node as a pair of coordinates in meters (floating point numbers). The coordinates can be arbitrary as long as they are non-negative. Nodes can only be deployed in the right upper quarter of the infinite Cartesian plane.

A radio-less node need not have its location specified (although it isn't forbidden). The default location of such a node is (0, 0).

The following <node> sub-elements are optional: <uart>, <pins>, <leds>, <eeprom>, <iflash>, <preinit>. They first five augment the nodes with optional components (like UARTs), which can be legitimately absent. The last one provides a way of initializing selected node attributes. All these elements are discussed below, each in a separate section.

5.4 The UART

Each node may be optionally equipped with a UART, whose functionality, as perceived by the praxis, accurately mimics two most popular ways of accessing the UART in PicOS: 1) direct interface (operations *ser_in*, *ser_inf*, *ser_out*, *ser_outf*, *ser_outb*), and 2) packet-style interface with the UART being viewed as a PHY for VNETI (TCV) (modes 'N' and 'L' described in *Serial.pdf*). This option is selected with the *mode* attribute of the <uart> element, which can be one of the following:

"direct"	(or "d")	selecting the default direct interface to the UART
"npacket"	(or "n")	selecting the 'N' mode interface over TCV
"ppacket"	(or "p")	selecting the 'P' mode interface over TCV

"line" (or "l") selecting the 'L' mode (line) interface over TCV

With the direct mode, the praxis can access the UART via the collection of "ser_" operations listed above. With the packet mode, the praxis can open the UART PHY by executing *phys_uart* (as described in *Serial.pdf*). Then it can associate a plugin with the UART PHY and use the interface essentially in the same way as an RF module. Finally, the line ('L') mode makes it possible to use the VNETI packet interface with a straightforward ASCII-oriented appearance of the UART at the other end.

The UART description in the input data assigns a bit rate to the UART, declares the sizes of two buffers to be used by the modeled driver, and determines what happens to the output and where the input will be coming from. The first three values are provided as attributes of <uart>, e.g.,

```
<uart rate="9600" bsize="12,8">
```

with the *rate* attribute being mandatory and *bsize* being optional. The first of the two *bsize* numbers gives the length of the input buffer,[7] and the second one declares the size of the output buffer. The default buffer size is 0 (in both cases), which effectively corresponds to "no buffer."[8] This is assumed for both buffers, if no *bsize* attribute is present in <uart>, or for the output buffer, if only one number is provided, e.g.,

```
<uart rate="9600" bsize="12">
```

The UART's interface to the real world is described by the <input> and <output> elements. The options are:

Local file or device:

```
<input source="device">device or file name</input>
<output target="device">device or file name</output>
```

The body of each element is stripped of any initial and trailing white spaces and the remainder is interpreted as a file name path relative to the directory where the model (the *side* program) has been invoked.

If the names in fact refer to files, than they should be different to make sense. On the other hand, they may represent the same device, e.g., a TTY terminal window. In that case, the UART may directly interact with the user.

The (input) characters arriving from a file/device, as well as those written by the praxis to the UART and sent to a file/device, are not preprocessed in any way (using the UNIX terminology, we would say that the interface is "raw"). The assumed interpretation of lines in PicOS, for the ASCII-oriented UART access functions *ser_out, ser_outf, ser_in, ser_inf,* is that an output line ends with CR+LF, and an input line must end with at least one of those characters, with any sequence of CR and/or LF characters at the end interpreted as a single end of line.

Both <input> and output elements accept an optional *coding* attribute, which can be "hex" or "ascii", with "ascii" being the default, e.g.,

```
<input source="device" coding="hex">
```

[7] This corresponds to a compilation parameter for PicOS.

[8] In fact a single-byte buffer is used by the model in that case, which corresponds to the hardware UART register on the microcontroller. This has the same meaning as the buffer size of zero in PicOS. Also, the MSP430 UART driver in PicOS uses 0 for the output buffer size (there is no parameter to change that.

With the “hex” coding, the input is assumed to be a sequence of bytes expressed as pairs of hexadecimal digits. Whenever a next byte is read from the UART, the emulator will look up in the file the next character looking like a hexadecimal digit (skipping all other characters). Then, if the following character is also a hexadecimal digit, the two will be decoded into a byte and returned as the read character. Otherwise, the program will be aborted with an error message. For output, the “hex” coding produces a sequence of 2-digit hexadecimal representations of the output bytes separated by spaces.

The <input> element accepts an optional *type* attribute, whose value can be “timed” or “untimed”, with the latter being the default, e.g.,

```
<input source=”device” type=”timed”>
```

For the “timed” type of input, the input file must be organized into records looking like this:

```
time { input data }
```

where time is a floating point number describing the time when the record will become available for input. If preceded by a '+' sign, the number describes the interval in seconds from the availability time of the previous record (or from 0 if the record starts the input sequence). If there is no '+', the number represents the absolute time in seconds from the beginning of run (time 0). If the time has already passed when the record is looked at by the system (i.e., the processing of previous records took more time than the record's availability time), the record becomes available immediately.

Once a record becomes available, its characters will arrive at the UART at the nominal rate specified in the <uart> element. If the praxis does not retrieve them on time, they will be lost. This is different from the “untimed” (default) operation whereby the bytes to be read by the praxis patiently await acceptance. In that case, the UART rate only determines how often they can be extracted (the minimum space between characters), but they never arrive faster than the praxis can cope with them.

Here is an example of a timed input sequence:

```
5.0 {s 4096\r\n} +4.0 {r\r\n} 3600 {s 0\r\n}
```

The string within braces may not include a closing brace unless escaped with a backslash. Generally, any character can be escaped with a backslash (including the backslash itself, which must be escaped). The escapes *\r*, *\n* and *\t* are treated as special characters (the last one standing for a TAB). An explicit newline also stands for itself, i.e., is equivalent to *\n*.

The timed mode can be combined with “hex” coding, e.g.,

```
5.0 {73 20 34 30 39 36 0d 0a} +4.0 {72 0d 0a} 3600 {73 20 30 0d 0a}
```

is equivalent to the previous sequence under “hex” coding.

Short input sequence specified directly in the data file:

The source attribute of <input> can be “string”, e.g.,

```
<input source=”string”>a direct sequence of bytes</input>
```

This specification is most useful in those circumstances when the node expects some short input from the UART at the beginning, e.g., to initialize the praxis. Generally, if the input part of the UART file is reasonably short, it can be inserted directly into the body of the <input> element, e.g.,

```
<input source="string">s 4096\r\n</input>
```

Note that the output may still be assigned independently to a file/device. Also, the “string” source can be combined with “hex” coding and/or “timed” mode, e.g.,

```
<input source="string" type="hex" mode="timed">
5.0 {73 20 34 30 39 36 0d 0a}
+4.0 {72 0d 0a}
3600 {73 20 30 0d 0a}
</input>
```

Defining a <default> UART whose input is mapped to a file is usually not a good idea, even if it works in principle. This is because, for a large network, the multiple instances of the file being opened for different nodes may deplete the population of allowable file descriptors and crash the model. On the other hand, having a default UART with an immediate input string (say for a default initialization of all the nodes) makes perfect sense. The output part of such an UART can be legitimately left unspecified.

Needless to say, the above ways of providing external input to a UART, or absorbing its output, are not very convenient (in fact they are practically useless) in the packet mode. Typically, in such a case, the praxis wants to talk to some OSS program implementing some non-trivial protocol. The proper way of handling this kind of interaction is to direct the modeled UART to an external agent (like udaemon – see below) providing a link between the virtual world of the VUE^2 model and the real-life OSS program.

Remote association through an agent:

The most flexible option is to map the UART to a socket and make it possible for external agents to claim its input and output. This is accomplished by using “socket” for the *source* and *target*, e.g.,

```
<input source="socket"></input>
<output target="socket"></output>
```

Once you map one end (say input) to a socket, the other end (output) also becomes mapped to a socket (there is no way to map it differently). Thus, the second line in the above sequence is redundant (although harmless). It may may be needed, if some other attribute of the element must be included, e.g., *coding="hex"*. A specification like this:

```
<input source="device">/dev/tty</input>
<output target="socket"></output>
```

will result in an error.

The body of an <input> or <output> element specifying “socket” is always ignored. External agents connect to such a UART following a special protocol that identifies the node (see Section 8.2). Thus, there is no need to provide any more details at this stage. A socket input can also be “hex” and/or “timed”. A socket output can be “hex”. Additionally, an <output> element specifying *target="socket"* may also include *type="held"* among its attributes, e.g.,

```
<output target="socket" type="held"></output>
```

The meaning of this setting is that the initial output written to the socket is saved and will be presented to the agent upon its (first) connection. This way you can make sure that whatever the praxis writes to the UART is never lost. This is important because before an agent can connect to the socket, the model must be started; thus, if the node writes something to the UART immediately after startup, that output might be lost.

Partially mapped UARTs

A socket UART is flexible in the sense that an agent may connect to it and disconnect many times. If anything is written to the UART while no agent is connected to it (excepting the period preceding the first connection to a "held" socket), that output is discarded and lost. Then if the node tries to read something from a disconnected socket, it will simply receive no data. This is the same as if the real UART were disconnected from the device.

For a device-mapped UART (and also for a UART whose input end is mapped to a string), it is legal to leave one end unmapped. For example, when writing to a UART whose output end is unmapped (but the UART is present), the output will be absorbed by the UART and discarded. An attempt to read from a UART with no input end will never return any data, but is formally legal. On the other hand, if no UART is defined at all, any attempt to reference it for I/O will trigger an error and abort the model.

You can indicate explicitly that the particular end is unmapped by saying something like this:

```
<output target="none"></output>
```

This is equivalent to simply skipping the specification. Note that a socket UART always defines two ends, even if only one end is explicitly mentioned in the declaration.

If a <node> element has no <uart> sub-element, but there is such a sub-element in <default>, then the <default> specification of UART will be assumed by the node. If the node wants to say explicitly that it has no UART, default or otherwise, the following declaration should be used:

```
<uart></uart>
```

This is the only variant of the <uart> element that requires no rate specification.

5.5 The LEDS module

For a node equipped with LEDs, you can make their status traceable or presentable to external agents. Here is a sample declaration of the LEDS module within <node>:

```
<leds number="4">
       <output target="device">led_status.txt</output>
</leds>
```

It says that the updates to the LEDs will be written to file *led_status.txt*. Each update takes one ASCII line beginning with the letter U (capital) and terminated with the newline character. The complete line looks like this:

```
U T F LLLLL ... LLL
```

where *T* is the time of the status change in seconds (a floating point number), *F* is 1 or 0, depending on whether the fast blink option is active or not (see PicOS operation *fastblink*), and each of the subsequent characters stands for the status of one LED from zero up (i.e., the length of the *L...L* string is equal to the number of LEDs). The values are: 0 – the LED is off, 1 – the LED is on, 2 – the LED is blinking. Here is an example:

```
U 10.325 0 0101
```

To make the LEDs status perceptible by external agents, use “socket” as the output target, e.g.,

```
     <leds number=″2″><output target=″socket″></output></leds>
```

Having connected to the LEDS module via the socket interface, the agent will be receiving updates in the same format as when they are written to a file.

Note that a LEDS module description can be placed in <defaults>. It may not make a lot of sense to direct LEDs status updates of all nodes to the same file, but it is perfectly legal to declare that all nodes can have their LEDs inspected by an agent (*target=”socket”*). If no agent is connected to the module, changes in the LEDs status trigger no external actions. Upon a connection, the agent will receive the current status of the LEDs, and then it will be receiving updates for as long as it is connected.

Similar to UART, to explicitly say that a node has no LEDs (overriding the default), you can use this declaration:

```
     <leds></leds>
```

Specifying zero as the number of LEDs has the same effect. In that case, the <output> specification is ignored if present.

5.6 The PINS module

This module provides an external interface to the node's I/O pins, including ADC and DAC. The node-inflicted changes to the (output) pins can be sent to an agent or stored in a file. Similarly, external changes to the pin voltage can be submitted by an agent or read from a file, possibly along with their timing. Here is a sample declaration:

```
     <pins total="10" adc="8" counter=″1,8,8″ notifier=″2,8,256″ dac=″6,7″>
           <output target="device">pins_output0.txt</output>
           <input source=″device″>pins_input0.txt</input>
           <status>1111111111</status>
           <values>0000000000</values>
           <voltage>0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0</voltage>
     </pins>
```

which is complex enough to illustrate all sub-elements that can contribute to <pins>. Except for the trivial declaration <pins></pins>, which explicitly states that the node has no PINS module, the *total* attribute of <pins> is required, and it specifies the total number of I/O pins in the module. If the *adc* attribute is present, it declares the number of pins capable of providing input to the analog-to-digital converter. That number cannot be larger than *total*. The ADC-capable pins constitute an initial subset of all pins.

The conventions assumed for identification and access to the pins are strongly related to the standard pin interface offered by PicOS (functions *pin_read, pin_write, pin_read_adc, pin_write_dac,* etc., including pulse monitor and notifier). Some familiarity with that interface will help you understand the ideas behind the PINS model.

If the PINS module is to be equipped with the pulse monitor, which consists of two pins: the counter and the notifier, the numbers of those pins can be specified with the optional *counter* and *notifier* attributes. The full specification of each of those pins takes up to three numbers. The first number identifies the pin, and must be less than the total number of pins, the remaining two values describe the “on” and “off” debouncing intervals expressed in milliseconds. Both, counter and notifier pins can be any pins, including those capable of ADC/DAC operation. Similar to PicOS, if the counter/notifier is not running (the function is switched off by the praxis), the respective pin can be used for its standard function.

The interpretation of the debouncing parameters is as follows. If the pin gets into the triggering state (depending on the edge setting, e.g., high for edge = 1), it must remain in this state for at least the “on” interval for the trigger to be considered valid. If the pin state changes within the “on” interval, the trigger is ignored. Similarly, once the trigger is successful, the pin must enter the opposite state for at least the “off” interval before it can be monitored for a next trigger. If a debouncing parameter is not specified (or is explicitly zero), the corresponding phase of the action is not debounced. Note that it is legal to specify a single debouncing parameter (which will be interpreted as the “on” interval, with the “off” interval assumed zero). If no debouncing parameters are specified, they are both assumed to be zero.

If any pins should provide DAC output, their numbers can be specified with *dac*. Note that only some models of MSP430 offer this functionality, which is confined to two pins.

The role of <status> is to indicate whether any of the pins in the range 0 ... *total*-1 are absent, i.e., the range includes holes.[9] The status of each pin is described by a single binary digit in the string, with 1 standing for “present” and 0 for “absent,” with the leftmost digit corresponding to pin 0. If no <status> element appears within <pins>, it is equivalent to “all ones,” i.e., no absent pins. If the string is shorter than the number of pins, the unaccounted for pins are all present by default.

The <values> element assigns initial digital values to all pins, which can be 0 or 1. Note that this assignment is only meaningful if the praxis sets the pin to input. At this stage, the declaration is equivalent to pulling the physical pin down or up via a resistor. Note, however, that it can be changed dynamically through an agent or from an input file. If the specification is absent, the pin values default to all zeros. If the string is shorter than the number of pins, the unaccounted for pins are all pulled down (initialized to 0).

Similarly, <voltages> assigns predefined analog values to the ADC-capable pins – in the natural order. When such a pin is selected by the praxis for its ADC function, this is the constant voltage that will show up on the pin for conversion. Again, that voltage should be viewed as initial as it can be changed by an agent of from an input file. If the specification is absent, the voltage defaults to all zeros. If the number of items in the list is less than the number of ADC-capable pins, the unaccounted for pins are set to voltage 0.

The <input> and <output> specifications are similar to those for UART, albeit simpler because the elements take no attributes besides “source” and “target.” The “string” source option is also available. There is a way to set up timed scripts for configurations of pin values via explicit sequences in the input data.

An input command addressed to a pins module is an ASCII line starting with one of the letters *T, P, D*. A *T* command requests a delay before reading the next command. The letter must be followed by a non-negative floating point number optionally preceded by a sign, e.g.,

```
T 12.5
T +0.01
```

In the first case, the number indicates the absolute time in seconds at which the next command should be read and interpreted. If the model is already past that time, the next command is read immediately. In the second case, the specified delay in seconds is calculated from the current moment.

[9] Even though the pin numbering is purely logical (defined on the per-board basis), the numbering convention assumed in PicOS makes it possible to exclude some pins from the continuous range.

A *P* command sets the digital value of a single pin. The letter is followed by the pin number in decimal followed in turn by 0 or 1, e.g.,

```
P 12 0
P 0 1
P 14 1
```

A *D* command assigns a voltage to the specified pin. The letter is followed by the pin number and the discretized voltage as a number from 0 to 32767 (0x7FFF) corresponding linearly to 0 ... 3.3V.[10]

An integer number can be specified in decimal (in the natural way) or in hexadecimal preceded with 0x. Initial spaces preceding the command letter are ignored. Thus, for example, the following sequence of commands makes sense:

```
T 0.5
       P 0xc 1
       P 0 0x0
       D 0xf 0x00ff
T 1.0
       D 0xf 0x0000
```

An immediate input string (the *source=”string”* option) should look like the contents of a file with a series of input commands, with the command lines separated with explicit new lines or *\n* sequences, e.g.,

```
<input source=”string”>P 1 1\nP 2 1\nT 0.01\nP 2 0</input>
```

is equivalent to:

```
<input source=”string”>
       P 1 1
       P 2 1
       T 0.01
       P 2 0
</input>
```

The first new line (the one preceding the first command) in the second version is stripped off by the data parser. This is not very important as empty lines in the command sequence are always ignored.

If the <output> part of the interface is configured and active, every change in the status of a pin results in a message being written to the file or sent to the agent. At the very beginning of the output sequence, there will be one special message looking like this:

```
N tt an
```

and specifying the number of pins: *tt* is the total number of pins and *an* is the number of ADC-capable pins (as specified in the attributes of the <pins> element). This line is also sent as a first message to any agent connecting to the PINS module over a socket – immediately after the connection has been established.

An actual update message starts with the letter U (capital) and consists of the following four components:

1. The time in seconds (with millisecond granularity) followed by a colon.
2. The number of the affected pin.

[10] No negative voltage is implemented at present (but probably will be later).

3. The status of this pin (a single decimal digit).
4. The pin value.

Here we have a few examples:

```
P 16 8
U 24.556: 4 1 0
U 3600.120: 7 3 176
U 2365.667: 2 0 0
```

The pin status value is interpreted as follows:

0 – digital input pin
1 – digital output pin
2 – an ADC pin
3 – DAC pin number 0
4 – DAC pin number 1
5 – pulse monitor counter pin
6 – pulse monitor notifier pin

The output value for cases 5 an 6 is always zero. For cases 2, 3, 4, the value is a number between 0 and 32767 representing the voltage (it is the output voltage for status 3 and 4, and the input voltage for status 2). For cases 0 and 1, the value can only be 0 or 1.

When the output is directed to a file, it starts with an *N* message followed by the complete list of initial pin values. From then on, the full status of all pins can be determined by tracking the updates reflecting changes in the individual pins. In the socket case, whenever an agent connects to the module, it immediately receives an *N* message followed by the initial series of “updates” for all pins reflecting their current status and values.

5.7 Buttons

PicOS offers a standard interface to buttons, which are selected pins capable of triggering “button pressed” events. In VUE^2, buttons can be declared in the input data set, within the context of <pins>, such that pertinent changes in their voltage will translate into button pressed events perceptible by the praxis (the *buttons_action* function). Consider this sample declaration:

```
<pins number="7">
       <input source="socket"></input>
       <buttons polarity="low" timing="200">0 2 5 6</buttons>
       <values>1111111</values>
</pins>
```

With this declaration, pins number 0, 2, 5, and 6 of the total 7 pins will be used as buttons (numbered 0 through 3). Their pressed status is low, and the single debounce parameter (200) stands for the minimum number of PicOS milliseconds separating two button press events. The list of debounce values can include two more numbers: the initial delay (in milliseconds) before an automatic repeat (if the button remains pressed) and the repeat interval (also in milliseconds). If the first of these numbers is zero (or not specified), auto repeat is disabled. If the second number is zero (or absent), it is assumed to be the same as the first number.

The legitimate values of *polarity* are *low*, *high*, *0*, *1* (alternatives for *low* and *high*), with high being the default. The numbering of buttons (as seen by the praxis) is always consecutive from zero up. The list of values in the body of <buttons> is interpreted as an array mapping indexes from zero up into pin numbers.

5.8 The SENSORS module

This module provides an external interface to the node's set of sensors and actuators, which are accessible by the praxis via the PicOS functions *read_sensor* and *write_actuator*. Here is a sample definition:

```
<sensors>
      <output target="device">actuator_output0.txt</output>
      <input source=”device”>sensor_input0.txt</input>
      <sensor number=”0” vsize=”2”>398</sensor>
      <sensor vsize=”1” delay=”0.001”></sensor>
      <sensor vsize=”3” delay=”0.1,0.3”>15</sensor>
      <sensor number=”4”>257</sensor>
      <actuator vsize=”2” delay=”1.0,2.0”>4095</actuator>
</sensors>
```

The <sensors> element has no attributes. The total number of sensors and/or actuators is determined from the element's contents and need not be announced. The arguments and definition layouts are the same for sensors and actuators.

All attributes of <sensor> and <actuator> are optional. The *number* attribute assigns a number to the sensor/actuator, which will identify it for the praxis (it corresponds to the second argument of *read_sensor/write_actuator*). If the number attribute is absent, the sensor/actuator will be assigned the next unused number. This is similar to the definition of nodes (Section 5.3). If the first sensor/actuator element has no number attribute, it is assumed to be zero. Note that sensors and actuators are numbered independently from zero. The maximum number for a sensor/actuator is 255.

The numbers of sensors/actuators defined in the data file need not cover a contiguous range, but each sensor/actuator can be defined at most once. Nonexistent sensor/actuators will trigger errors when referenced. Note that sensor number 3 in the above example does not exist, but sensor 4 is there. Holes like this may make little sense, but they are not illegal.

Attribute *vsize* determines the size of the sensor/actuator value in bytes. Legal sizes are 1, 2, 3, and 4. The default size is 4, but, if the definition specifies the range (see below), and *vsize* is absent, the size will be determined from the range: as the smallest number of bytes needed to accommodate the maximum possible value stored in the sensor/actuator. In all cases, the value to be extracted from a sensor or stored in an actuator is interpreted as an unsigned integer, which is never negative.

Size 3 is discouraged and may never be used for a real sensor/actuator. While sizes 2 and 4 are interpreted as *word* and *lword* types in an endian-independent way, it is impossible to do the same for a 3-byte number. For now, such a number is interpreted as little-endian.

The *delay* attribute should consist of one or two (comma separated) floating point numbers. It describes the delay in seconds incurred in reading a value from the sensor (via *read_sensor*) or storing a value into the actuator (via *write_actuator*). If there are two numbers, then the delay will be generated as a random value between them. Note that the second number must not be less than the first one. If the *delay* attribute is absent, operations on the sensor/actuator incur no delays.

If the body of a <sensor> or <actuator> element contains a strictly positive integer number, that number is interpreted as the range, i.e., the maximum value that the sensor/actuator is able to assume. The minimum is always zero. If no range is specified, it is inferred from *vsize* as the maximum unsigned value that can be stored on the specified number of bytes. If neither *vsize* nor the range is present, then, by default, the size is assumed to be 4 and the range is 2^{32}-1.

The <input> and <output> specifications are exactly the same as for PINS. If the input is from a socket, then the output is implicitly assumed to be present and directed to the same socket. As for all other types of objects, It is illegal to associate one direction with a socket and the other with something else.

An input command addressed to the module is an ASCII line starting with one of the letters *T* or *S.* Similar to PINS, a *T* command requests a delay before reading the next command. An *S* command has this syntax:

```
S sn v
```

where *sn* is the sensor number and *v* its new value. The rules for encoding numbers are as for PINS. Here is a sample sequence of commands:

```
T 0.5
S 0 223
S 1 0xffe
T +1.0
S 0x2 19999
```

If the <output> part of the interface is configured and active, every change in the value of a an actuator results in a message being written to the file or sent to the agent over the socket. If the connection is over a socket (as opposed to a device), then, additionally, every change in the value of a sensor results in a similar message. The latter can be used by the remote agent as an acknowledgment of its last *S* request.

The first message sent Immediately after initiating the connection, or after node reset, looks like this:

```
N an sn
```

and tells the number of actuators (*an*) and sensors (*sn*) defined for the node. Strictly speaking, accounting for the possible holes, *an*/*sn* is the maximum number of a defined actuator/sensor + 1. This is followed by the list of initial “update” messages, each looking like this:

```
A an vvvvvvvv mmmmmmmm
```

or this:

```
S sn vvvvvvvv mmmmmmmm
```

The second message type is only present for a socket interface. The messages tell the current (initial) values of all defined actuators/sensors together with their ranges. The first number following the letter (*A* or *S*) is the decimal sensor/actuator number (between 0 and 255 inclusively), the next (hexadecimal) number is the value, and the last (also hexadecimal) number is the maximum. All actuators appear before sensors, and both types of objects are listed in the increasing order of their numbers; thus, a hole directly corresponds to a sensor/actuator that is absent.

Following the initial “update,” subsequent updates are sent without the range component, i.e., the letter (*A* or *S*) is only followed by two numbers. The second number is always in hexadecimal and consists of exactly eight (hexadecimal) digits (no *0x* prefix).

5.9 Storage modules: EEPROM, SD, information flash

VUEE allows you to model two kinds of external storage: generic EEPROM (which may also be used to mimic an SD card) and the so-called *information flash* (e.g., corresponding to the

Flash Information Memory available on MSP430 processors). Both types (i.e., EEPROM and information flash) can be present simultaneously. The list of operations applicable to them can be found in the *Storage.pdf* document (coming with PicOS). Operations on SD cards are mapped to the corresponding operations on EEPROM. Note that you cannot have them at the same time (as at most one EEPROM module can be defined per node). There is no access to general code flash in VUEE (operations *cf_write*, *cf_erase*). These operations are completely void in VUEE: *ee_open*, *ee_close*, *ee_panic*, *sd_open*, *sd_close*, *sd_panic*, *sd_idle*. Those of them that return values behave as if they always succeeded.

EEPROM is declared with the <eeprom> element (see Illustration 1). The attributes of that element specify the parameters of the module, while the body may contain initializers (<chunk> elements) that pre-load values into specific fragments of the storage. Here is the list of attributes of <eeprom>:

size=”*n*” or **size=**”*n,m*” (where *n* and *m* are integer numbers)

This is the only one attribute that is mandatory, in the sense that its absence means that EEPROM is undefined. A possible application of such a declaration, e.g.,

<eeprom></eeprom>

is to indicate that the node should have no EEPROM even if there is one in <defaults>. Note that no partial inheritance from <defaults> takes place, i.e., if an EEPROM module is defined at a node at all, then all its parameters must be specified with that definition (the absent ones assume global default values that need not match those in <defaults>).
The first number of *size*, *n*, specifies the total size of the storage in bytes. The second (optional) number, *m*, gives the number of pages (or blocks). This is important if you want to make sure that the erase operation applies to an entire number of pages/blocks, in which case *m* provides the requisite grain. If the number of pages, *m*, is present, then the total size *n* must divide evenly by *m* and the result (the page size) must be a power of two.

clean=”*xx*”

The value is a single-byte hexadecimal number which specifies the contents of an erased byte. If the attribute is missing, the value defaults to **FF**.

erase=”byte” or **erase=”page”**

The attribute specifies the granularity of erase, i.e., whether you can erase individual bytes (this is the default) or pages. The latter case only makes a difference when the page size is nontrivial.

overwrite=”yes” or **overwrite=”no”**

The attribute indicates whether overwriting an already written byte (without a prior erase) produces the correct result (the first case, which also happens to be the default). In the second case, it is assumed that overwriting zeros with ones has no effect.

timing=”*rl,ru,wl,wu,el,eu,sl,su*”

This attribute specifies the timing parameters that determine the delays incurred by operations *ee_read*, *ee_write*, *ee_erase*, *ee_sync.* For those operations that admit a state argument (and can possibly block), i.e., all of the above except for *ee_read*, those parameters determine the amount of time elapsing until the respective operation unblocks after its start. This will translate into an actual delay modeled by VUE^2. For *ee_read* (which has no state argument and never blocks, VUE^2 cannot simulate the delay. This also happens when the state argument of *ee_write*, *ee_erase*, or *ee_sync* is **WNONE**. In that case, the delay is only applied to calculating the power usage by the power tracker (see below) and otherwise ignored. In particular, SD card operations (which do not accept the state argument) behave like the corresponding EEPROM operations with the state argument being **WNONE**.

The timing parameters are floating point numbers providing the lower and upper limits (in seconds) for the four operations (in the listed order) per one byte, except for ee_sync (which takes no length argument). The delay in each particular case is determined as a uniformly-distributed random number between the bounds. If not all numbers have been specified, the missing ones (from the end) are assumed to be zeros, which has the effect of not modeling any delays for the corresponding operation(s).

image=”*filename*”

This attribute says that there is a file backup for the storage. If *filename* refers to an existing file, this file will be opened and its contents will appear as the initial contents of the storage. If the file is larger than the formal size of the storage, it will be truncated down to size. If it is shorter (in particular, if its length is zero or it doesn't exists), it will be extended and filled up with “erased” bytes (depending on the *clean* attribute). Whenever some bytes are written to the EEPROM, those bytes will be stored in the file, which can be subsequently viewed, archived, and so on after the experiment.

It is not illegal to use the same *filename* for multiple nodes and may make perfect sense if the storage is used read-only. This is not checked, however, so you will mess things up if multiple nodes write to the same backup file. As a precaution, it is formally illegal to use this attribute if the <eeprom> element occurs in <defaults>.

EEPROM can be preinitialized with a sequence of <chunk> elements appearing within the body of the <eeprom> element (see Illustration 1). One mandatory attribute of such an element is *address* providing the offset into the EEPROM where the specified chunk of bytes goes. The bytes themselves can be listed as the body of <chunk>. For example, this chunk:

```
<chunk address=”8192”>
      0 1 2 3 FE 0xAC 0 0 0 11
</chunk>
```

will preset 10 bytes starting at address 8192 with the specified contents. The numbers appearing within the body of a <chunk> are all in hexadecimal (the **0x** prefix is optional) and each of them stands for exactly one byte.

Another possibility is to read the contents of a file into an EEPROM fragment, e.g.,

```
<chunk address=”0” file=”fonts.nok”></chunk>
```

In this case, the body of the <chunk> element is ignored, and the entire contents of the specified file are written into the storage at the specified address.

A single <eeprom> definition may include multiple chunks of different kinds. They can overlap (being applied in the order of their occurrence), but none of them is allowed to extend beyond the formal limits of the storage. An attempt to do that will be signaled as an error.

Note that initialization with chunks is compatible with backing files. For example, this definition is legal:

```
<eeprom size=”536870912,1048576”
    image=”myimage.bin” clean=”00” overwrite=”no”>
      <chunk address=”0”>ba ca de ad”</chunk>
      <chunk address=”16384” file=”picture.nok”>
</eeprom>
```

and its outcome is intuitively clear. The contents of the initializing chunks (wherever they come from) will be written into the backing file.

Note that EEPROM (and information flash), unlike other node components, are not re-initialized after the (modeled) reset.

Functionally, the information flash is a subset of EEPROM. The <iflash> element used for its definition accepts the same attributes as <eeprom> (with the same meaning), except for *erase*, *overwrite*, and *timing*. Even though *clean* is admissible, *overwrite=”no”* is forced for <iflash>, as is *erase=”page”*. Similar to EEPROM, information flash can be initialized (the body of <iflash> may include <chunk> elements) and/or backed to a file.

5.10 Nokia LCD display N6100P

An <lcdg> element appearing as part of <node> (or <defaults>) equips the node with the model of a graphic LCD. The *type* attribute of this element determines the display model. At present, the only legitimate type is *n6100p*; thus,

```
<lcdg type=”n6100p”></lcdg>
```

incorporates a virtual N6100P into the node. The body of the <lcdg> element should be empty. A sequence without the type attribute, i.e.,

```
<lcdg></lcdg>
```

indicates that the LCD model is absent and can be used to nullify locally a global declaration appearing in <defaults>.

5.11 Power tracker

The power tracker module allows you to maintain a record of energy usage by a node and, in particular, estimate the average amount of current drawn by the node within a certain time period. At present, the built in functionality of the power tracker covers: the CPU states (low power versus normal), the RF module, non-volatile storage (EEPROM, SD card), and sensors.

Power tracker parameters are specified on a per-node basis (see page 25) and may appear as part of a <node> element or in <defaults> (with the standard meaning). Here is a sample specification involving all four components that are traceable at present:

```
<ptracker>
    <output target="socket"/>
    <module id="cpu">0.3 0.0077</module>
    <module id="radio">0.0004 16.0 30.7 30.7</module>
    <module id="storage">
         0.030 0.030 10.0 15.0 17.0 16.0
    </module>
    <module id="sensors">0.0 2.5</module>
</ptracker>
```

The specification can include an <input> and/or an <output> element with the standard capabilities, e.g., as for SENSORS or PINS. Similar to SENSORS, an input command addressed to the module is an ASCII line. If the first character of such a line is *T*, then the line describes the timing of the command in the next line, similar to SENSORS or PINS. Otherwise, that character can only be *C* (and any remaining characters in the line are ignored). This command clears (i.e., zeroes) the tracker effectively starting a new measurement at the current moment.

The <module> elements describe (intentionally) the current usage of the respective components (modules) in their different states. The meaning of those states depends on the component. The first state (and the first number) has a standard meaning representing the default (reset) state of the module. For example, value 0.3 at the CPU <module> can be interpreted as 0.3mA drawn by the board's CPU in the default idle state assumed after reset. Note that the interpretation of all those numbers is up to the user, and they have no implied standard meaning.

The second value at the CPU <module> represents the power down state (7.7uA). This component uses only two values.

For the RF (radio) component (the second module), we have 4 numbers describing, respectively, the current drain in the off state, with the receiver on, with the transmitter on, and with both the receiver and the transmitter simultaneously on (if it makes a difference). Generally, if the expected sequence of values is longer than the one provided, all the unspecified values are assumed to be the same as the last one. Thus, the last value for the radio <module> is redundant.

The storage module applies to EEPROM. The values respectively refer to the amount of current drawn by the module:

- in the off (or closed) state (e.g., after executing *ee_close*)
- in the on state (e.g., after *ee_open*) but otherwise idle
- while reading
- while writing
- while erasing
- while syncing

The amount of time needed for the module to remain in an active state (reading, writing, erasing, syncing) is determined from the timing parameters (page 41). If there are no timing parameters (for all or some operations), the corresponding current drain is not modeled (assumed to be zero).

For the sensors module, the second and possibly subsequent values refer to the current drain by the sensors or actuators (in the order of their numbers) while active. Note that sensors and actuators admit timing parameters (page 39). In the above example, the single “on” value (2.5mA) will apply to all sensors and actuators.

Not all modules have to be specified. A missing module uses no energy: its current drain is always zero regardless of the state.

If the output target of the power tracker is “device”, i.e., the updates regarding current drain are sent to a file, VUE^2 will be writing to that file a message after every change in the state of any of the tracked components. Such a message looks like this:

```
U T [E]: A V
```

where ***T*** is the time of the state change in seconds (from the beginning of run), ***E*** is the elapsed time of the current measurement (counted from the last moment the tracker was zeroed), ***A*** is the accumulated average current drain from the beginning of the measurement (since ***E*** seconds ago), and ***V*** is the current drain at the moment (after the state change). All four numbers are floating point (the last two may include exponents).

If the output target is "socket", a line in the above format is sent to the remote agent at 1 second intervals for as long as the connection is on.

5.12 Pre-initialization

By pre-initialization we understand assigning initial values to some node attributes in a way that is conceptually different from the normal (dynamic or static) initialization in the node program. Intentionally, it corresponds to burning some characteristic values into the node's flash memory, e.g., serial numbers, that make the node unconditionally distinguishable from other nodes.

Pre-initialized values are represented in the input data as <preinit> elements, which can be sub-elements of <node> or <default>. Here is an example:

```
<preinit tag="ESN" type="lword">0x8000ff01</preinit>
```

The mandatory *tag* attribute assigns a name to the preinit. The second (optional) *type* attribute determines the object type, which can be:

lword	a 32-bit long integer value (signed or unsigned)
word	a 16-bit integer value (signed or unsigned)
string	a character string

These indicators can be abbreviated down to the initial letter. If the type attribute is absent, it defaults to *word*.

The element's body contains the value, which should agree with the type specification. For types *lword* and *word*, the value should be a decimal number (possibly signed) or a hexadecimal value (beginning with *0x*). Regardless of its actual size, it will be truncated to the respective size of the object. For the *string* type, the body is simply viewed as a character string.

The way to reference a preinit in the praxis code is via this function (which is a method of *PicOSNode* – see page 9):

```
IPointer preinit (const char *tag);
```

which returns the preinit value represented by the *tag*. The generic output type must be cast to the proper object type.

The most natural place to put calls to the above function is file *attribs_init.h* (see page 13), which contains “static” initialization of node attributes, e.g.,

```
_da (ESN) = (lword) preinit (“ESN”);
_da (Greeting) = (char*) preinit (“WELCOME”);
```

If the tag referenced by *preinit* is not found among the <preinit> elements associated with the current node, the function consults the preinits of <defaults>. If it is not found there, the function returns zero, i.e., the object is initialized to zero (or to a NULL string pointer). This corresponds to leaving the object uninitialized in the real world.

6 Mobility drivers

The underlying SMURPH/SIDE vehicle allows the programmer to create arbitrarily elaborate mobility models as external SMURPH processes. VUE^2 provides an interface to external daemons that can modify coordinates of nodes by sending commands over sockets (Section 9.5). The same interface can be fed from files (or devices).

6.1 Declaring a mobility driver

A mobility driver accepting node positioning commands[11] is not a node module (like UART, LEDS or PINS), because its domain is not restricted to a single node: it may affect the positions of multiple nodes essentially at the same time. One such driver, capable of receiving connections from remote agents over sockets, is available all the time and need not be declared in the input data file. We shall call it the *external* driver. It is also possible to declare *local* mobility drivers, which can modify node positions according to a local description, i.e., included directly in the input data or read from local files. Such a driver accepts a subset of the commands available to a remote agent connected to the external driver

A local mobility driver is declared with the <roamer> element (see Illustration 1), which is a sub-element of <network> and looks like this:

```
<roamer>
    <input source=”...”> ... </input>
</roamer>
```

where the input source specification is essentially the same as for PINS (Section 5.6). A <roamer> element with <input source=”socket”> is ignored. Note that there is no <output> sub-element in <roamer>. While an external agent connected to a mobility driver over a socket can query the driver and receive feedback information (see Section 8.6), this functionality is not available to a local driver.

An arbitrary number of local mobility drivers can be declared in the same data file. Each of them can affect the position of an arbitrary number of nodes. Once a driver requests to re-position a node, any active positioning request regarding that node, possibly originating from another driver, is canceled and the new request takes over.

6.2 The commands

Essentially there are three types of commands accepted by a local mobility driver. These commands constitute a subset of requests available to external agents talking to a mobility driver over a socket. The simplest command simply assigns new coordinates to a given node and has this format:

```
M nn x y
```

where *nn* is a node number (according to the numeration of <node> elements), and *x* and *y* are the new planar coordinates. The operation amounts to teleporting the node to the new location. The move is instantaneous. Note that the coordinates cannot be negative.

Similar to PINS (page 36), it is possible to delay the interpretation of subsequent requests with a *T* command, e.g.,

```
T +0.01
```

will cause a 10 ms delay before interpreting the next request, while

[11] Strictly speaking, positioning commands apply to Transceivers rather than nodes.

```
T 3600
```

will halt the input until the model has been run for one hour since its startup.

With a sequence of appropriately timed (tiny) teleportations one can implement any movement, closely approaching smooth navigation along any curve at any (possibly variable) speed. It is tedious, however, to procure such data sets by hand. To simplify typical experiments, the mobility driver offers a standard random-way-point roaming model, which is invoked with the following command:

```
R nn x0 y0 x1 y1 smin smax pmin pmax duration
```

where all numbers except *nn* (the node number) are floating points. When such an request is accepted, the indicated node starts roaming an continues doing so for *duration* seconds according to the pattern prescribed by the remaining parameters. If *duration* is zero or negative, the roaming continues forever, or rather, until another repositioning request is issued to the node. In the meantime, the driver is free process other commands: the roaming is carried out in the background.

The first four floating point numbers, i.e., *x0, y0, x1, y1,* describe the rectangle bounding the node's roaming area. Note that *x0* must not be greater than *x1,* and *y0* must not be greater than *y1.* Besides, all coordinates must not be negative. The node will travel according to this algorithm:

1. A uniformly distributed random target location is generated within the bounding rectangle as well as a uniformly distributed random speed between the specified minimum (*smin*) and maximum (*smax*) in meters per second. The node moves at this speed towards the target location.

2. Upon reaching the target location, a uniformly distributed random pause time is generated between *pmin* and *pmax* seconds. The node rests for this much time and proceeds from 1.

This process continues for *duration* seconds, or indefinitely (if *duration* is zero or negative). It will be interrupted when the same or different mobility driver issues any re-positioning command to the node.

Taking advantage of the *string* source for a roamer, you can easily set up multiple random-way-point mobility scenarios for multiple nodes. For illustration, consider this declaration:

```
<roamer>
    <input source=”string”>
      R 0 [0.0 200.0 0.0 200.0] [3.5 10.0] [1.0 10.0] -1
      R 1 [0.0 200.0 0.0 200.0] [3.5 10.0] [1.0 10.0] -1
      T 1800.0
      R 5 [0.0 200.0 0.0 200.0] [3.5 10.0] [1.0 10.0] -1
      R 7 [0.0 200.0 0.0 200.0] [3.5 10.0] [1.0 10.0] 7200.0
      T +1800.0
      M 5 40.0 50.0
    </input>
</roamer>
```

Nodes 0 and 1 start roaming immediately at the beginning of the experiment. After 30 minutes, nodes 5 and 7 join them. Node 5 stops roaming after another 30 minutes when it is teleported to location (40.0, 50.0) and becomes stationary. Node 7 keeps moving for two hours before stopping (at some random location), while nodes 0 and 1 roam forever.

The above example suggests that the command syntax allows for non-numerical characters to be interspersed among the numbers (we used square braces for clarity). This only applies before floating point numbers and should be viewed as an undocumented feature that may be removed without warning. The maximum length of a single request line is 112 characters, which means that extravagance is not encouraged.

7 Panels

By default, the program (praxis) running at a node is automatically started as soon as the input data file is read and processed, at the very beginning of the model's execution. It is possible to have some nodes stopped initially by specifying *start="off"* in their <node> elements (see page 24). You may even have all nodes stopped initially (by putting start="off" as an attribute of <defaults>) and then start them later, e.g., at specific moments. Once started and running, a node can be stopped, started, and reset an arbitrary number of times. This process can be described in the input data (or a separate file); it can also be controlled by external agents.

7.1 Declaring panels

The respective data element, called <panel>, is illustrated in the sample data set on page 25. Its full format is similar to that of <roamer> (Section 6) and looks like this:

```
<panel>
    <input source=”...”> ... </input>
</panel>
```

where the input source specification is essentially the same as for PINS (Section 5.6). A <panel> element with <input source="socket"> is ignored. Similar to <roamer>, there is no <output> sub-element in <panel>. While an external agent connected to a panel driver over a socket can control the status of nodes and receive feedback information (see Section 8.7), this functionality is not available to a local driver. An arbitrary number of <panel> elements can be present in the same data file. Each of them can affect the status of arbitrary nodes. For example, a node stopped by one panel can be restarted by another. Stopping a stopped node or starting a running node has no effect. Resetting a node always starts it up, regardless of whether it was stopped or running before the reset.

7.2 Commands

The simple set of commands understood by a panel consists of the timing request *T* – as for PINS (page 36) and three status change requests (in all cases, *nn* is the node number):

O *nn* - to switch the node on
F *nn* - to switch the node off
R *nn* - to reset the node

For illustration consider the following sequence:

```
T 30.0
O 1
T +5.0
O 2
T +10.0
F 1
F 2
R 3
```

Thirty seconds after the beginning of execution node 1 will be switched on, then 5 seconds later, node 2 will be switched on, and after 10 more seconds, nodes 1 and 2 will be switched off and node 3 will be reset.

8 The agent protocol

A running VUE^2 model makes a socket available for connections from agents. Such agents may implement GUI to various station components (UART, LEDS, PINS, ROAMER) or provide an interface to OSS programs for the target praxis (executed on a real network). For example, an agent may implement an open ended UART driver (like a COM port under Windows) connected to a node running under VUE^2.

The standard port number of the socket opened by a VUE^2 model is 4443. It can be changed via the *port* attribute of <network> (see Section 5), e.g.,

```
<network nodes="48" port="5590">
```

which can be useful, e.g., if multiple VUE^2 models are run on the same system.

At present, VUE^2 implements eight functions (i.e., connection types) for agents. They are:

UART	for connecting to a node's UART
LEDS	for connecting to a node's LEDs module
PINS	for connecting to a node's pins module
CLOCK	for reading the virtual clock of the model
ROAMER	to affect the locations of nodes
PANEL	to control the on/off status of nodes
SENSORS	to receive values from virtual sensors and affect virtual actuators
LCDG	to send data to the virtual graphic LCD
PTRACKER	to receive updates regarding energy consumption by nodes

Except for ROAMER and PANEL, there can be multiple active instances of any connection type at any given moment – referring to the respective components of different nodes. For example, several UARTs belonging to different nodes can be interfaced to agents simultaneously.

In all cases, the interface is session-oriented, meaning that an agent does not connect to the model for a single inquiry, but sets up a session during which it will be involved in an exchange of potentially unlimited length. The connection can be broken by the agent at any time by simply disconnecting.

8.1 The handshake

A connection is initiated by the agent (client) connecting to the agent port of the VUE^2 model for a TCP (stream) session. Immediately after receiving a connection, the agent should send a polling sequence shown in Illustration 2.

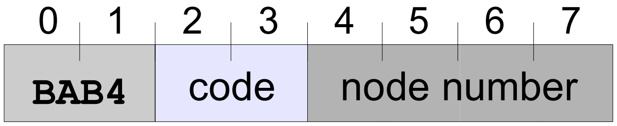


Illustration 2: Connection polling sequence

It consists of 8 bytes interpreted as two short numbers and one long number in the network format, i.e., MSB first. The first number is a magic sequence used for a quick assessment of the request's sanity. The second short number, *code*, describes the requested service type. For a request related to a specific node (e.g., connection to a UART), the last four bytes specify the node number, according to the numeration of <node> elements in the input file (Section 5.3).

Having received a polling sequence, the VUE^2 program responds with four bytes (a 32-bit unsigned number in network format) interpreted as an acknowledgment. The least significant byte of that number indicates a success or failure. If its value is 129 (decimal), it means that the request has been accepted. The upper three bytes may return additional information to the connecting program. At present, only the UART module uses those bytes to pass the UART's bit rate as the number of bits per second divided by 100. Thus, an acknowledgment for UART connection has the format shown in Illustration 3. For the remaining modules, bytes 0-2 contain zeros.

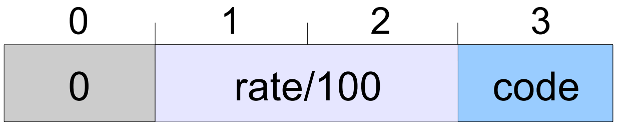


Illustration 3: UART connection response

If the acknowledgment code is different from 129, it means an error. Having sent the error code, the VUE^2 program immediately closes its end of the connection. Here is the list of possible error values:

0 wrong magic sequence
1 node number out of range
2 illegal (unimplemented) request code
3 some agent is already connected to this particular module
4 timeout; this is only sent if a complete 8-byte polling sequence does not arrive within 30 seconds from the moment of connection
5 the module (UART, PINS, LEDS, ...) is present at the node, but it has no socket interface
6 this code is sent when the program discovers that the other side has closed the connection, before closing its end; thus, it is unlikely to be ever received
7 the request sequence is too long
8 illegal request or query
128 the node has no such module (in response to a connection request)

8.2 UART protocol (request code 1)

This kind of request requires a valid node number. If the number is OK, and the indicated node is equipped with a UART module with socket interface, and no other agent is connected to that UART at the time, the request is accepted. Then, following the acknowledgment word sent by the VUE^2 program, the connection becomes entirely dedicated to the UART. This means that whatever the agent sends over the socket will appear as input on the UART, and whatever the praxis writes to the UART will be sent to the agent over the socket. This will continue until the session is torn down (closed) by the agent (or until the VUE^2 program terminates).

The format of the data sent/received by the UART conforms to the coding and type attributes associated with the UART in the input data (Section 5.4). In particular, if the input is “timed”, the data must be organized into packages preceded by the playback time. Normally, this kind of operation is not very useful for a socket connection, but it is available. Note that when the playback time is in the future, the input will be blocked until that time. Similarly, with “hex” encoding of the respective end, the data follows the hexadecimal format described in Section 5.4.

8.3 PINS protocol (request code 2)

This request requires a valid node number. If the number is OK, and the indicated node is equipped with a PINS module with socket interface, and no other agent is currently connected to the module, then the request is accepted. Following the acknowledgment word, the VUE^2 program immediately (without polling by the agent) sends an *N* message (indicating the number of pins – see page 37) followed by the full list of updates reflecting the current status of all pins. Each such a message takes a complete ASCII line of text, as described in Section 5.6, terminated with a single newline character. The agent may assume that the messages arrive in the order of pin numbers from zero up. Note that the total number of pins and the number of ADC-capable pins are included with every message.

No other message types ever arrive from the VUE^2 end. Whenever the status of a pin is changed by the praxis, a pertinent message is queued for transmission. Such a message always refers to a single pin.

The agent is able to affect the status of pins by sending messages over its end of the socket that look exactly like those described in Section 5.6. Such a message should be an ASCII string ending with a single newline character. With an on-line agent connection (as opposed to reading the pin status from a file), there is little demand on *T* requests, although they are not prohibited. Note that if such a message specifies a moment in the future, the input will be blocked until that time.

8.4 SENSORS protocol (request code 7)

This request requires a valid node number. If the number is OK, and the indicated node is equipped with a SENSORS module with socket interface, and no other agent is currently connected to the module, then the request is accepted. Following the acknowledgment word, the VUE^2 program immediately (without polling by the agent) sends an *N* message (indicating the number of actuators and sensors – see page 40) followed by the full list of updates reflecting the current values and ranges of all actuators and sensors. Each such a message takes a complete ASCII line of text, as described in Section 5.8, terminated with a single newline character.

No other message types ever arrive from the VUE^2 end. Whenever the value of a sensor/actuator is changed by the agent (in the first case) or by the praxis (in the second), a pertinent message is queued for transmission. Such a message always refers to a single sensor/actuator. Note that only the initial updates carry information about the ranges (Section 5.8).

The agent is able to affect the values of sensors (not actuators) by sending messages over its end of the socket that look exactly like those described in Section 5.8. Such a message should be an ASCII string ending with a single newline character.

8.5 LEDS protocol (request code 3)

This kind of connection works one way, i.e., following the initial 8-byte polling sequence, the agent never sends anything to the VUE^2 program. Immediately after the acknowledgment word, the VUE^2 program sends to the agent the initial configuration of LEDs as a message described in Section 5.5. This is an ASCII message terminated with a single newline character. Then, a similar message is sent whenever the status of a LED changes.

Additionally, to be able to detect the disappearance of the agent, the program sends a dummy NOP message, which is simply a single newline character, every 10 seconds, unless there is a LED status change to report. Note that no NOP messages are sent when the LEDS module is interfaced to a file.

8.6 ROAMER protocol (request code 4)

This type of connection requires no node number (which is ignored, but must be present in the polling sequence). By following this protocol, the agent is able to carry out the operations described in Section 6.2, as well as receive feedback from VUE^2 regarding node positions. Only one remote ROAMER connection can be active at any time. Following the acknowledgment word sent by the VUE^2 program to the agent, the rest of the conversation is in ASCII, with lines terminated by (single) newline characters.

Immediately following the confirmation byte, the VUE^2 program awaits requests from the agent. The agent may query the program for the position of a given node with a command that consists of the letter *Q* followed by a single nonnegative node number, e.g.,

```
Q 49
```

In response, the program will send the following line:

```
P nn total x y name
```

starting with the letter *P* and consisting of two integer numbers in hexadecimal (*nn, total*), two floating point numbers (*x, y*) and a string (*name*). The first number is the node number and should be the same as the number sent in the query. The second number gives the total number of nodes in the network. The two floating point numbers are the node coordinates in meters. The string is the node's type name, i.e., the name of the SMURPH class representing the node. The latter is useful when the model consists of nodes of several types, as it allows the agent (with an appropriate set of queries) to recognize the complete configuration of the network. If the agent knows nothing, it can always safely issue a query for node 0. Then, having learned the total number of nodes, it can poll them all for position and type.

The agent may issue at will the commands described in Section 6.2. Also, it will receive an update whenever the position of a node changes, including changes incurred by the agent itself. Note that local mobility drivers may be changing things behind the scenes. VUE^2 tries to reduce the number of unnecessarily updates to avoid overflowing the connection with traffic under heavy mobility scenarios.

An update message begins with the letter *U* and looks like this:

```
U nn x y
```

where *nn* is the node number and *x* and *y* are its new coordinates.

Remember that node coordinates cannot be negative. Having received an invalid request, i.e., one specifying an illegal node number or a negative coordinate, the program sends error code 12 to the agent (a single byte) and closes the connection.

8.7 PANEL protocol (request code 5)

Similar to ROAMER, this type of connection requires no node number (which is ignored, but must be present in the polling sequence). By following this protocol, the agent is able to carry out the operations described in Section 7.2, as well as receive feedback regarding node status. Only one remote PANEL connection can be active at any time. Following the acknowledgment word sent by the VUE^2 program to the agent, the rest of the conversation is in ASCII, with lines terminated by (single) newline characters.

Immediately following the confirmation byte, the VUE^2 program awaits requests from the agent. In addition to the commands described in Section 7.2, one more (query) request is available to the remote agent, which looks like this:

Q *nn*

where *nn* is a node number. If *nn* is a legitimate node number, the program sends in response a line in the following format:

nn s total name

where *nn* is the node number specified in the original request, *s* is a single character O (the node is on) or F (the node is off), total is the total number of all nodes in the network (i.e., the limit for the range of legitimate node numbers), and *name* is the type name of the node, i.e., the name of the SMURPH class representing the node. Typically, an agent connecting to this service will begin its conversation with a query for node 0 (which is always safe) after which it will learn the total number of nodes.

In addition to being able to issue the commands described in Section 7.2, the agent will also receive an update whenever the status of a node changes (e.g., due to a panel described in the input data set – see Section 7), including changes incurred by the agent itself. Such an update message looks like a query response (see above), except that in includes only the first two items, i.e., the node number and its status. Note that when an active node is reset, its status does not change, so such an operation performed by a data-driven panel will be unnoticed by the agent.

Having received an invalid request/query, i.e., one specifying an illegal node number, the program sends error code 12 to the agent (a single byte) and closes the connection.

8.8 CLOCK protocol (request code 6)

With this simple protocol, the agent can receive time information from the VUE^2 program. Following the acknowledgment word, the program will be sending at second intervals the number of virtual seconds from the beginning of execution. This number arrives as a four-byte binary integer in the network format. Every message consists of exactly four bytes.

8.9 LCDG protocol (request code 8)

This is a one-way protocol with data being sent solely to the display. Following the acknowledgment word, the program is sending display updates consisting of binary short (16-bit) words in the network format. The most significant 4-bit nibble of a word determines the type of information carried in it:

0 means that the remaining 12 bits of the word contain the value of a pixel (4 bits per pixel: RGB, from most to least significant); the pixel is to be stored in the “current” position on the screen (as explained below)

1 means that remaining 12 bits of the word contain a command (as explained below)

8-15 i.e., the most significant bit being set, means that the word contains a repeat count for the last pixel; whatever last pixel value was received from the emulator, it must be repeated (*w* & 0x7fff) + 1 times (in addition to the original first appearance), where *w* is the word content

Here are the command codes:

0x300	(i.e., *w* = 0x1300); four words following this one contain the bounding rectangle of an area to be updated (filled) with subsequently arriving pixels; the rectangle is specified as the ending value of X, the starting value of Y, and the ending value of Y (the coordinate of the left upper corner of the display is 0, 0
0x400	end of update; this command means that a logically complete series of updates has been concluded and, for example, it now makes sense to present a new updated variant of the display to the user
0x100	switch the display on
0x200	switch the display off

In addition to defining a bounding rectangle, command 0x300 resets the current pixel position to the left upper corner of the rectangle, i.e., the starting X and Y coordinates. Whenever a new pixel (including auto replications) is stored, the X coordinate is advanced until it exceeds the ending coordinate, in which case it wraps back to the starting X coordinate, while the Y coordinate is incremented by 1. When the Y coordinate exceeds its ending value, it wraps back to the starting value, and so on.

8.10 PTRACKER protocol (request code 9)

Following the initial 8-byte polling sequence, the agent will be receiving from the simulator a message, as described in Section 5.11, every second or whenever the state of one of the monitored component changes, whichever comes first. This is an ASCII line terminated with a single newline character. When the agent wants to zero the tracker (i.e., start the measurement from scratch), it should send back a simple command consisting of the letter ***c*** followed by a newline.

9 UDAEMON

Here we briefly describe the functionality of udaemon, which is a Tk program implementing a rudimentary GUI agent for conversing with a VUE^2 model in execution. The program is started with an optional argument, which indicates the port number used by VUE^2 program. By default, this number is 4443 and agrees with the VUE^2 default (Section 8).

Note: the present version of udaemon requires Tcl/Tk version 8.5 or higher. The program may work correctly with older versions of Tcl/Tk, but the U-U feature (see below) will not work with version 8.4.

When started, the program opens a window that looks as shown in Illustration 4. This window is the launcher of five different interfaces, corresponding to the five protocols described in Section 8.

The text area labeled “Node Id” is used to insert the (decimal) node number, for those interfaces that require it. The “select” button, initially showing “UART (ascii)*,”* offers the menu shown in Illustration 5. Having selected a given interface (and, if required, inserted a node number in the text area), you can click the *Connect* button to bring up the respective interface window.

The area below the buttons displays various textual messages (the log), which, in particular may include error messages. The area is scrollable and stores the last 1024 lines of the log (which practically always means everything).

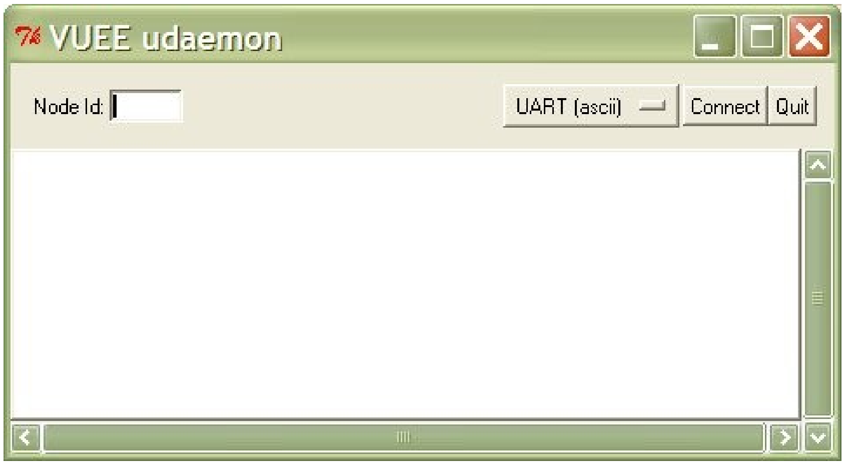


Illustration 4: The root window of udaemon.

9.1 The UART interface

Note that the UART interface occurs in two versions on the menu: “ascii” and “hex.” These versions are independent of the “ascii/hex” coding discussed in Section 5.4. In this place, “hex” means hexadecimal display of the data arriving from the UART, as well as hexadecimal entry of the data to be sent to the UART, and is useful in those cases when that data are non-ASCII (e.g., the praxis is meant to communicate with some program using binary data, and we want to manually emulate the behavior of that program). A similar effect will be achieved when the UART coding is “hex” (and the interface mode at udaemon is “ascii”); however, in that case, the hexadecimal coding/decoding will be done by the VUE^2 program. Note that double “hex,” i.e., on both sides, will have a rather confusing effect of displaying in hexadecimal the character codes of hexadecimal digits (the input will be even more confusing).

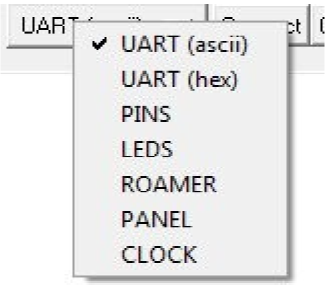
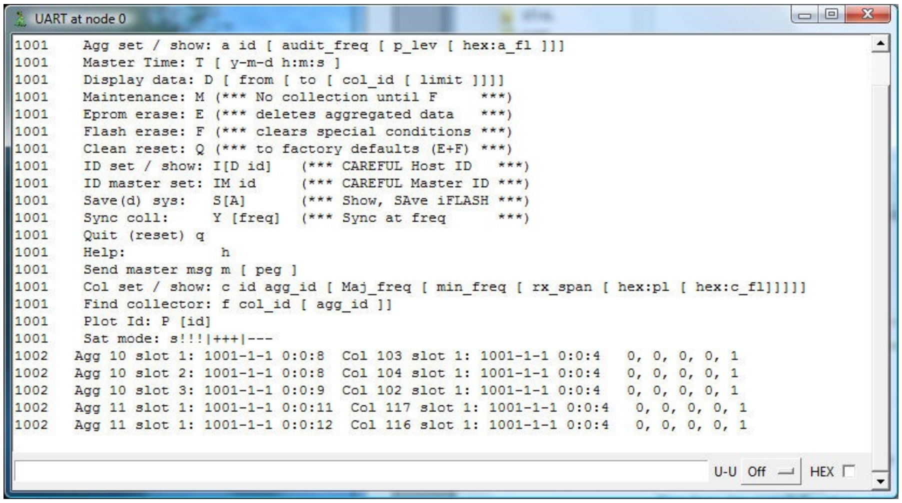


Illustration 5: The interface menu.

A UART connection produces a window shown in Illustration 6. It looks like a straightforward terminal emulator, with the text area at the bottom used for input. The upper area is resizeable and scrollable. It stores the last 1024 lines written to the UART.

By checking the “hex” box at the bottom, you can switch the terminal from “ascii” to “hex” and vice versa without reconnecting. The reason why, in addition to this box, the mode can also be selected from the root window is that when the UART window is open, some text may already be displayed on it (see the held option on page 33). The mode switch only affects the data to be displayed, not that already present on the screen.
A line typed into the bottom text area is sent to the VUE^2 program when you hit the *Enter* key. For “hex” mode of the UART terminal, this data should consist of pairs of hexadecimal digits optionally separated by spaces. Only the binary data represented by those digits will be sent to the VUE^2 program.



```
1001     Agg set / show: a id [ audit_freq [ p_lev [ hex:a_fl ]]]
1001     Master Time: T [ y-m-d h:m:s ]
1001     Display data: D [ from [ to [ col_id [ limit ]]]]
1001     Maintenance: M (*** No collection until F     ***)
1001     Eprom erase: E (*** deletes aggregated data   ***)
1001     Flash erase: F (*** clears special conditions ***)
1001     Clean reset: Q (*** to factory defaults (E+F) ***)
1001     ID set / show: I[D id]   (*** CAREFUL Host ID   ***)
1001     ID master set: IM id     (*** CAREFUL Master ID ***)
1001     Save(d) sys:   S[A]      (*** Show, SAve iFLASH ***)
1001     Sync coll:     Y [freq]  (*** Sync at freq      ***)
1001     Quit (reset) q
1001     Help:           h
1001     Send master msg m [ peg ]
1001     Col set / show: c id agg_id [ Maj_freq [ min_freq [ rx_span [ hex:pl [ hex:c_fl]]]]]
1001     Find collector: f col_id [ agg_id ]]
1001     Plot Id: P [id]
1001     Sat mode: s!!!|+++|---
1002    Agg 10 slot 1: 1001-1-1 0:0:8  Col 103 slot 1: 1001-1-1 0:0:4   0, 0, 0, 0, 1
1002    Agg 10 slot 2: 1001-1-1 0:0:8  Col 104 slot 1: 1001-1-1 0:0:4   0, 0, 0, 0, 1
1002    Agg 10 slot 3: 1001-1-1 0:0:9  Col 102 slot 1: 1001-1-1 0:0:4   0, 0, 0, 0, 1
1002    Agg 11 slot 1: 1001-1-1 0:0:11  Col 117 slot 1: 1001-1-1 0:0:4   0, 0, 0, 0, 1
1002    Agg 11 slot 1: 1001-1-1 0:0:12  Col 116 slot 1: 1001-1-1 0:0:4   0, 0, 0, 0, 1
```

Illustration 6: A UART window.

The U-U button (whose default state is Off) allows you to interface the UART to a real serial port (which may be a COM port on Windows or a tty device on Unix). For that, press the button and select one of the devices available on the pop-up list. On Windows, entries looking like CNCB0, CNCB1, and so on, correspond to the B-ends of COM-to-COM port pairs emulating null modems. The requisite driver (for Windows) can be downloaded from http://com0com.sourceforge.net/.[12] This way, you can have real-life OSS programs talk to PicOS praxes executed under VUEE (with the OSS program talking to the A-end of the respective pair).

On Linux, the same feature can be implemented using pseudo-ttys, i.e., devices named /dev/pts/*n*, where *n* is a number. In directory PICOS/Linux (of the PicOS package, not VUEE), you will find a file named *nullmodem.c*, which compiles trivially, e.g.,

```
gcc -o nm nullmodem.c
```

When you run the resulting executable, it will create two pseudo-ttys, show them to you, e.g.,

```
TTY name: /dev/pts/7
TTY name: /dev/pts/6
```

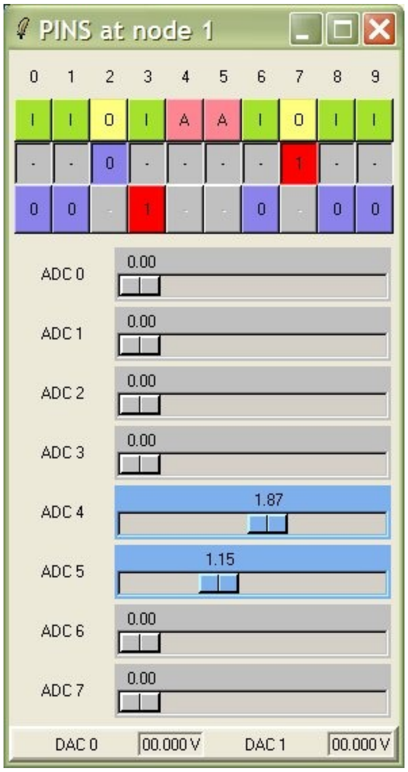
[12] This package is also available from http://www.adremnetworks.com/olsonet/, files com0com-...-i386-fre.zip (32-bit version) and com0com-...-x64-fre.zip (64-bit version). For best effects, having installed the driver and set up a pair of ports, say CNCA0 and CNCB0 (which are created by default), invoke the *setupg* utility and select “emulate baud rate” and “enable buffer overrun” for both ends.

and, from that point on, act as a forwarder between them. One of those pseudo-ttys should then be selected from the U-U menu of the UART window, while the other can be opened by the OSS program. The formal baud rates used at the two ends don't have to match.

When a virtual UART of udaemon is connected to a real serial device, the UART window mirrors the exchange between the node and the program attached to the other end of the real port. The input field also works in that mode (your entries will be interleaved with any data arriving from the real UART), as does the HEX checkbox.

9.2 The PINS interface

Similar to UART, this interface requires a valid node number. Following a successful connection, a window like the one shown in Illustration 7 pops up on the screen.



0	1	2	3	4	5	6	7	8	9
I	I	O	I	A	A	I	O	I	I
-	-	0	-	-	-	-	1	-	-
0	0	-	1	-	-	0	-	0	0

ADC 0 0.00
ADC 1 0.00
ADC 2 0.00
ADC 3 0.00
ADC 4 1.87
ADC 5 1.15
ADC 6 0.00
ADC 7 0.00

DAC 0 00.000 V DAC 1 00.000 V

Illustration 7: A PINS window.

The three rows of boxes at the top reflect the current status of the pins, with one column corresponding to one pin. Only the bottom row is clickable, and only if the letter in the upmost box of the column is *I*, which means that the pin has been set by the praxis to “input.” For such a pin, clicking on the box in the bottom row toggles the binary value of the pin. In Illustration 7, pins 0, 1, 3, 6, 8, 9 are input, pins 2 and 7 are output (letter *O*), and pins 4 and 5 are set for analog input (letter *A*). The full collection of characters that can appear in a bottom-row box is:

I	the pin is set for digital input
O	the pin is set for digital output
A	the pin is set for analog input
P	the pin is a pulse counter
N	the pin is a notifier

D the pin is a DAC output
– the pin is unused (its status field in the input data set is 0, see page 35)

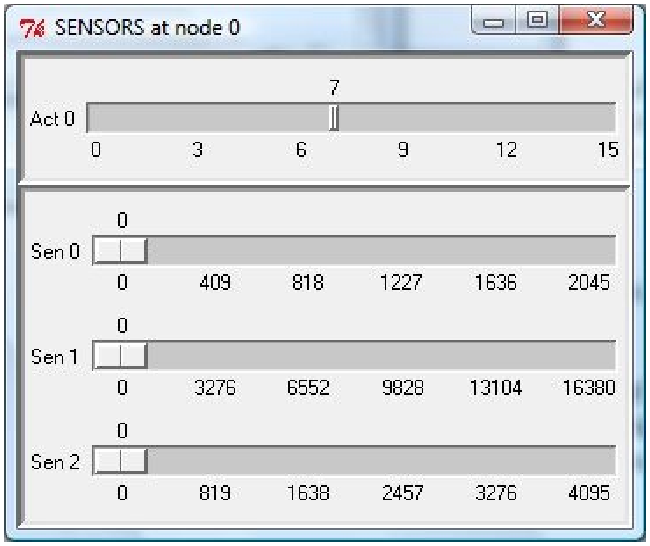
The middle row shows the output value of those pins that have been set by the praxis as "output." For any other pin type, the corresponding square contains a dash. Those squares in the bottom row that do not represent input pins are disabled, i.e., clicking on them triggers no action.

A right click on a clickable pixel square has the effect of two consecutive clicks separated by a 300 ms interval. The role of this feature is to emulate pressing a button (Section 5.7).

Each ADC-capable pin has a slide widget that can be used to set the voltage on that pin. Only those pins that have been currently selected for the ADC function have their slides enabled (pins 4 and 5 in Illustration 7). The value above the slide knob tells the current voltage on the pin. Finally, the DAC-capable pins have voltage display areas at the bottom. No such pins appear to be available in Illustration 7.

Note that the praxis can dynamically redefine pins, and, for those that are output or DAC, set their values. All such changes are immediately reflected in the window. You can trigger changes in those pin values that are currently available for digital input or ADC – by clicking on a box in the bottom row or adjusting the respective slide. Such changes are immediately conveyed to the VUE^2 program. Slide adjustments are sent incrementally, meaning that a slow movement of the slide knob will result in several updates sent to the VUE^2 program, reflecting the progress in adjustment.

9.3 The SENSORS interface



SENSORS at node 0

Act 0 7

Sen 0 0

Sen 1 0

Sen 2 0

Illustration 8: A SENSORS window.

Similar to PINS, this interface requires a valid node number. Following a successful connection, a window like the one shown in Illustration 8 appears on the screen. The upper section displays the values of the node's actuators. There is no way to affect an actuator value from the window.

Every sensor defined at the node has a slider in the lower portion of the window. By adjusting the sliders, you can modify the sensor values. An update is sent when the mouse is released.

9.4 The LEDS interface

This interface is very simple as it involves no user input. The displayed window shows one circle for each of the LEDs defined in the module, with the LEDs numbered from left to right. Color gray means that the LED is off, otherwise, it is on. The first five LEDs show the colors: red, green, yellow, orange and blue when lit. If there are more than five LEDs, the ones numbered 5 and up are all red.

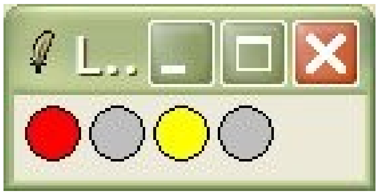


Illustration 9: A LEDS window.

9.5 The ROAMER interface

This interface takes no node number. Illustration 10 shows a sample window displayed in response to a ROAMER connection, for a network consisting of 6 nodes of the same type. The numbers in the lower right corner show the dimensions of the area covered by the window.

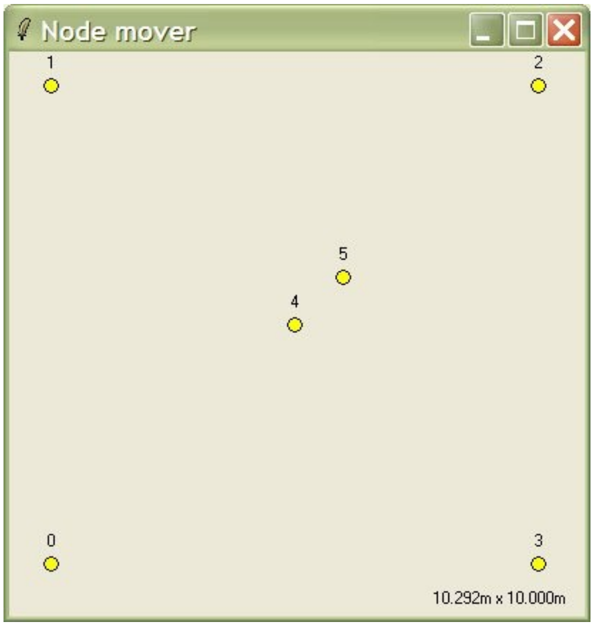


Illustration 10: A ROAMER window.

If the window is resized, udaemon will re-fit the nodes to the new size, i.e., by extending the window, you do not necessarily make it cover a larger area, but simply magnify the picture. It isn't difficult to guess that by dragging a node, you will move it to a new location. If the dragging is smooth, multiple requests will be sent to the VUE^2 program, such that the movement will appear incremental to the model. Note that the present version of udaemon has no interface for specifying random-way-point mobility patterns (see page 47).

Moving to the right and up is unrestricted. If you move a node beyond the window boundary, the window will be renormalized to the new network geometry. This renormalization does not affect the window's size or shape on the screen, but the assumed dimensions of the edges. Moving to the left and down is limited by the coordinates <0,0>. Udaemon will not let you move a node below these coordinates.

To see the exact position of a node, just move the mouse over it. The coordinates will be displayed in the lower left corner of the window along with the node's type name, which coincides with the name of the node's class. If there are multiple node types in the model, they will receive different colors on the screen – up to six colors: yellow, blue, orange, red, green, gray. If there are more than six node types, all the remaining types will be gray.

There is a way to see the distance between a selected pair of nodes. Click on the first node without dragging it, then move the mouse over the second node and click. A dashed line connecting the nodes will show up for 2 seconds with a number in the middle showing the distance between the nodes in meters.

9.6 The PANEL interface

This interface takes no node number. Illustration 11 Shows a sample panel window. Immediately after the request is issued, the window only displays the status of node 0. More nodes can be added to the window by entering the node number in the text area at the bottom and pressing the *Add* button. Nodes can also be removed from the panel by pressing the blue *Delete* button on the right.

As shown in Illustration 11, nodes 0, 1, 3-5, and 7 are on, while nodes 2 and 6 are off (halted). The on status of a node is indicated by the red background color of the node's label. For the three status change buttons, color yellow means that the button is enabled. Note that Reset is always enabled: resetting an off node will also start it up and make active.

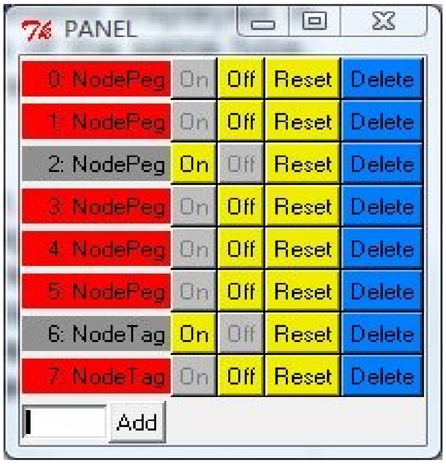

PANEL
0: NodePeg On Off Reset Delete
1: NodePeg On Off Reset Delete
2: NodePeg On Off Reset Delete
3: NodePeg On Off Reset Delete
4: NodePeg On Off Reset Delete
5: NodePeg On Off Reset Delete
6: NodeTag On Off Reset Delete
7: NodeTag On Off Reset Delete
Add

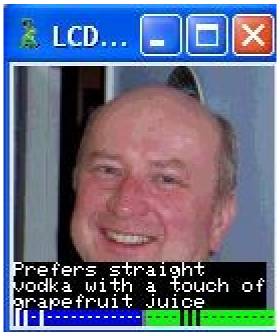
Illustration 11: A panel window.

9.7 TIMER interface

This is a very simple interface that needs no node number. It opens a window that shows the emulated time of the model in seconds, assuming that the execution started at time 0.

9.8 LCDG interface

This is a simple passive (reception only) interface presenting a 130x130 pixel display looking as shown in Illustration 12.

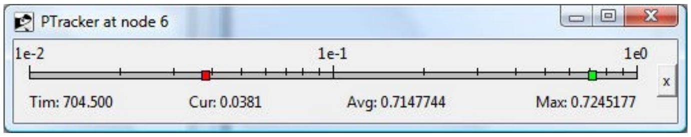


Prefers straight
vodka with a touch of
grapefruit juice

Illustration 12: LCDG model

9.9 PTRACKER interface

This is a single window looking like the one in Illustration 13.



Tim: 704.500 Cur: 0.0381 Avg: 0.7147744 Max: 0.7245177

Illustration 13: PTRACKER window

The red bar shows the current current drain (note that the scale is logarithmic), while the green bar shows the accumulated average. The small button on the right is used to zero the power tracker.

9.10 Forcing specific window positions on the screen

Sometimes, e.g., for a demonstration, you may want to organize the multiple windows on the screen, such that they will show up in specific, predefined, fixed locations. This may reduce the confusion resulting from multiple windows of the same type (e.g., UART, PANEL) piling up in almost the same (or some random) area. Udaemon accepts an optional input file which allows you to predefine the locations of selected windows. The default expected name of that file is *geometry.xml*. If a file with this name is present in the current directory when the program is invoked, it will use that file automatically. A different file can be specified explicitly with the *-G* option, e.g.,

```
udaemon -G /home/pawel/MyStuff/position.def
```

The file consists of simple XML statements. Here is a sample content:

```
<geometry>
      <roamer>  = (100,230) </roamer>
      <uart> 0,1,6-9,15 = (300,250) </uart>
      <clock> = (-170,-45) </clock>
</geometry>
```

All the declarations appear as sub-elements of the single <geometry> element. Elements corresponding to the particular window types are named after the respective windows (e.g., on the interface menu) in all lower-case. Here is the full list of legitimate names: *uart*, *clock*, *leds*, *lcdg*, *pins*, *sensors*, *panel*, *ptracker*, *roamer*, and *root*, with the last one representing the root window (see Illustration 4).

The minimum acceptable syntax for the position associated with a particular window (which is provided as the text body of the respective XML element) consists of = followed by two numbers in parentheses (white spaces are ignored). The first number specifies the x-coordinate (pixel number) of the left edge of the window, and the second number gives the y-coordinate (pixel number) of the upper edge. A negative number (like in the <clock> element in the above example) represents a negative offset (in pixels) from the maximum coordinate, i.e., from the right/lower boundary of the screen.

If = is preceded by a list of numbers, then those numbers refer to the nodes to which the window may apply (this only makes sense for those windows that may exist in multiple versions indexed by the node number). For example, the <uart> element from the above example will be applied to the UART windows for nodes 0, 1, 6, 7, 8, 9, and 15. All those windows will be displayed in the same place, which probably makes little sense, unless, for some reason, only one of them is displayed at a time. The primary purpose of this feature is to differentiate among different versions of a window for different nodes. For example, in the most straightforward case, you may do something like this:

```
...
<leds> 0 = ( 10,200) </leds>
<leds> 1 = (120,200) </leds>
<leds> 2 = (230,200) </leds>
<leds> 3 = (340,200) </leds>
...
```

The <roamer> element allows for two additional (and optional) attributes, e.g.,

```
<roamer image="imgfile.jpg" width="1200.0">=(10,20)</roamer>
```

which can be used to specify a background image for the roamer's canvas. That file must be in one of the formats acceptable by Tk (the Img package), e.g., *jpeg* files are OK. The path to the image file will be interpreted from the directory in which udaemon has been called. Attribute *width* associates a width with that file, i.e., specifies the extent of the image's width in the geographical units (meters) used by VUEE to interpret node coordinates. The corresponding height is derived from the ratio of the image's height to its width.

With a background image defined, the roamer's window becomes inflexible and cannot be resized. The program assumes that the network area covers a rectangle of the specified width (and the derived height) starting at point (0,0). Some narrow margins within that area (stripes at the bottom, top, left, and right) are reserved and cannot be used for node coordinates. If a node coordinate falls outside the network rectangle, udaemon will complain (with a pertinent message in the root window) and display the node at the window's boundary (such that all nodes are always accounted for). In contrast to the standard version of the window, nodes cannot be moved (with the mouse) outside the boundary of the fixed network rectangle. Note that they can still be moved by the simulator (if it runs a mobility model), in which case they will be pushed to the window boundary.

Instead of *width*, you can specify *height* (in that case the width will be derived). As long as the image attribute is present, at least one of them is required. You can also specify both width and height, which (if they don't agree with the image dimensions) will override the image scaling. Note that the text field of an element specifying a background image for the roamer can be empty, i.e., you don't have to specify a predefined location for the window.