PAWiS Simulation Framework Overview

Johann Glaser and Daniel Weber

July 1, 2008

Abstract

The PAWiS Simulation Framework facilitates the simulation of wireless sensor networks. This includes the internal structure of the sensor nodes as well as the network connecting them. This overview document describes the basic principles of the framework and gives a short introduction of its structure. A large part of this document is taken by a tutorial including coding examples which guide you to the usage of the framework.

1 Wireless Sensor Networks

Wireless sensor networks (WSN) are mainly used in building automation, car interior devices, container tracking, bridge and vulcan maintenance and monitoring as well as geological surveillance. The nodes comprise sensors for e.g. temperature, humidity, insolation, strain gauge and so on.

All nodes communicate with each other via an *ad-hoc multi-hop* network. This means that the packets are routed from node to node and the routing information accrues during the network operation. Special MAC and routing protocols were developed for this purpose. The probably best known is IEEE 802.15.4 ZigBee ([H+03], [IEE03]), but even more specialized protocols exist (e.g. CSMA-MPS [MB04]).

The main development target is the low power consumption of every node to supply them with energy scavenging or from a single battery, yet with long lifetimes as long as 10 years.

2 Simulation Framework

The PAWiS Simulation Framework assists you in the development and especially the optimization of WSN nodes and network protocols. The internal structure of the nodes as well as the communication between them are simulated. The wide range of utilisation starts at tiny sensor nodes (e.g. the TinyMote [Roe04]) and reaches via tire pressure monitoring and car climate control to as complex systems as home entertainment (e.g. Sindrion [GSS+04]).

The internal structure of a node is built as a *virtual prototype*. This means that its function, the timing and power consumption as well as communication and node failures are simulated. With the true top down development methodology (see Fig. 1) the design starts at a functional specification and implementation. Guided by requirements the design is refined via architectural models down to models reflecting the actual implementation. On the other hand, physical constraints effect the functional and architectural designs from bottom up.

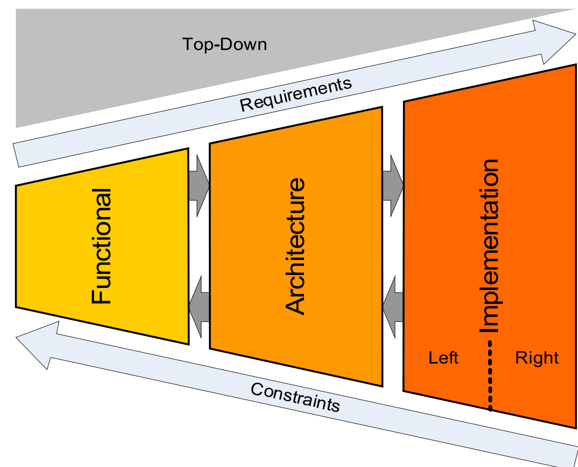

Figure 1: Top down design methodology.

3 Work flow

The WSN node is split into *functional blocks* like a physical, a MAC, a routing and an application layer, a CPU, serial interface, AD converters, timer and so on. Each of these blocks is then mapped to modules in the simulation. For every functional block its *module type* is defined. This comprises its name and the standardized interfaces. For each of the module types several different implementations can be prepared.

The *work flow* of the design and optimization of a WSN with the PAWiS Simulation Framework is depicted in Fig. 2. At the beginning the node is composed from one module implementation per module type from the module implemen-

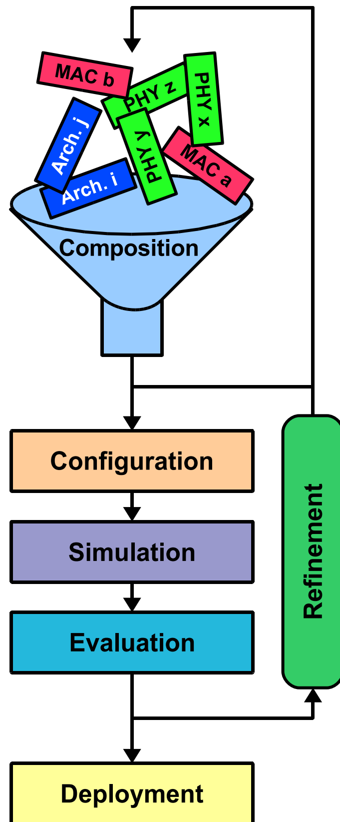

Figure 2: Refinement cycles.

tations library. Then the modules are configured (e.g., clock frequency of the CPU, resolution of the ADC). In the next step the model is simulated and the results are evaluated.

Based on the simulation results the individual models are refined, i.e. their behaviour is implemented in more detail and more accurately. Additionally the modules' functions are altered to approach to the design goals. The configuration of the models may be modified too and then the simulation is performed again. Alternatively the composition of the total node may be changed, if the current selection of module implementations does not meet the design goals.

These cyclic improvements of the models are called *refinement cycles* and are the main track to enhance the development. When these refinement cycles are completed, the final outcome comprises

- the function and
- the architecture of the node, as well as
- the implementation and
- the power specification of every sub-module.

4 Optimization

Several strategies for the optimization of the wireless sensor system are available. First of all a *system level optimization* is performed. This includes the node composition and even modifications of the whole system behavior like choosing different network layout or application patterns. The system level optimization guides you to an adequate system architecture.

In parallel *cross-layer optimization* is performed. This means that more than one network layer is modified at a time. Probably each of these changes alone would degrade the node performance, but the interaction of them leads to an improvement of the total node.

All this optimization is performed by applying the following strategies.

- Exchange the actual module *implementation* for one module type, i.e. a different selection from the library, e.g. a dual-slope, a $\Sigma\Delta$ or an SAR ADC. Another example is choosing different MAC protocols.
- *Partitioning* of modules and/or functions by dividing the task between hardware and software, digital and analog or RF and baseband. For example a specific MAC protocol could be implemented in software or as a dedicated hardware acceleration unit. A combination of both is also possible.
- The *scale* of a module, e.g. the resolution of an ADC or the register count of a CPU.
- *Parameterization* of modules, e.g. the timing, transmission power and bit rate of a radio transmitter.

5 The Framework

5.1 Structure

In Fig. 3 the basic structure and dependencies of the PAWiS Framework are depicted. The framework is based on the OMNeT++ discrete event simulation system and the C++ programming language. The model programmer mostly interacts with the framework and C++.

The Framework requires the following tools: OMNeT++ discrete event simulator 3.3 and Doxygen 1.4.7. To compile on Unix machines you need the GCC compiler 4.0 or 4.1, Autoconf 2.60a and Automake 1.9.6. On Windows the Microsoft Visual Studio version 7 or 8 are required. Please consult the installation guide located at `https://clara.tuwien.ac.at/pawis/sim:doc`.

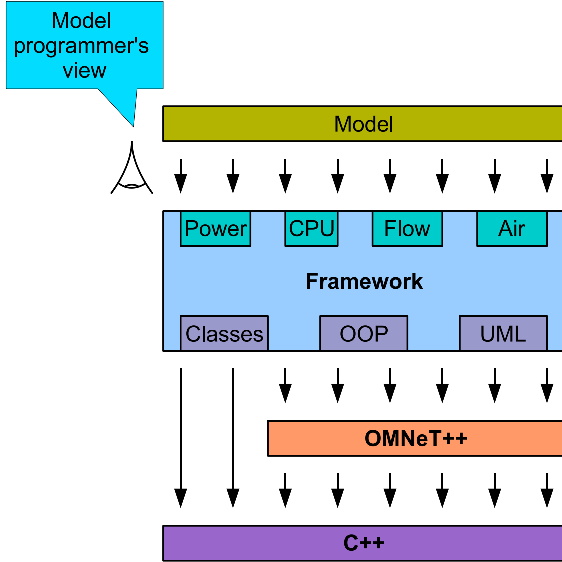

Figure 3: Structure of the PAWiS Framework.

5.2 Discrete Event Simulator

Since the PAWiS Simulation Framework is based on the OMNeT++ Descrete Event Simulation System, we have to discuss how its event processing is working. Basically every module can generate new *events* which are stored in the so called *Future Event List* (FEL). The simulation kernel steadily picks the earliest event (which has the semantic of a message transported from network node to node) from this FEL. The current simulation time, which is stored in a variable, is then set to this event's time and the event is delivered to the destination module.

The destination module may itself insert new events into the FEL. As soon as this task has finished, the simulation kernel again consults the FEL and takes the earliest event.

The simulation time therefore does not correlate to wall clock time, because it is only defined by the events generated during simulation. For simulations with large temporal distance between the events, simulation time will elapse faster than the wall clock time. Contrary, simulations with dense temporal resolution, where the simulation kernel has to execute many events and their handlers, the simulation time may be slower than the wall clock time.

5.3 Basic Concepts

5.3.1 Modularization

The wireless sensor node is split up into modules, e.g. a CPU, the network protocol layers (application, routing, MAC, physical), a timer and so on. Each of these modules is implemented as a C++ class derived from `PawisModule`.

5.3.2 Tasks

Within one module several tasks can be implemented and even run in parallel. The execution within one task is sequential but all tasks are running concurrently. Every task is implemented as one class method.

This concurrency is implemented as cooperative multithreading, so every task has to define points where it passes execution to the simulation kernel. That means that the program flow is stopped at this point and later continues there. At these yield points the simulation time elapses. Everything between two such yield points happens in the same instant of simulation time.

5.3.3 Functional Interfaces

Control flow transitions between two modules are implemented as so called *Functional Interfaces*. These are similar to subroutine calls (see the green arrows in Fig. 4). Every functional interface is implemented as a task, so it can run in parallel to the other tasks. Note that the invocation of a functional interface starts this task and after the task has finished, the invocation returns. That means that functional interfaces are not running permanently but only when invoked.

Additionally at some points a module can define a wait condition depending on another module. This is complemented by a mechanism to trigger a parallel task (see red arrows in Fig. 4).

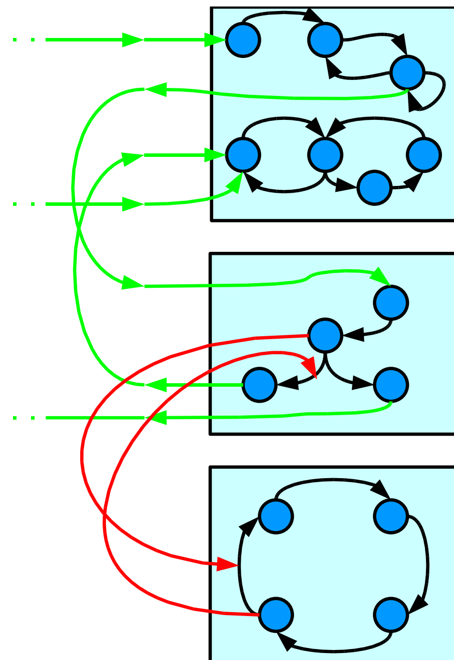

Figure 4: Tasks inside modules and control flow across module borders.

5.3.4 CPU

The submodules of a sensor node are usually either implemented as *firmware* (software), i.e. are executed by a CPU, or as *dedicated hardware*. Every module can have several tasks, and the can be mixed software and hardware tasks (e.g. the RF transceiver hardware plus its driver). It is important to note that software tasks can not run in parallel, since the CPU is only available once. Therefore the control flow is transferred from one module to another (think of subroutine "call" and "return" here) instead of triggered events.

To model the power and time consumption of software tasks the ideal case is a CPU simulator which is fed by the actual firmware opcodes. Unfortunately this is very time consuming. Therefore in the PAWiS framework the CPU simulation is split into two parts.

The functional tasks of the firmware are written in pure C++ code inside of the task method itself. The timing and power consumption part is then "delegated" to the CPU module. It only reports its power consumption (on behalf of the real module) and delays execution for the estimated processing time. This delegation is realized as `requireCpu(`*percentage integer*`,` *percentage float*`,` *percentage memory access*`,` *percentage flow control*`,` *duration*`)`. This call is one of the exit points of task coroutines.

These requests include the estimated execution time of the firmware code on the CPU. Now think that the CPU of a given node should be replaced during the optimization process (e.g., use an ARM instead of an MSP430) or change the configuration (e.g., use a faster crystal). This would also require to modify all execution time estimates in all modules of the node. To allows for a CPU exchange without the need to adapt other modules the execution time estimates are referred to the so called *norm CPU*. This is an imaginary but well defined CPU implementation (regarding its performance). The actual CPU model scales its processing time and power consumption according to its individual properties and depending on the percentages of execution types.

The framework user has to implement the CPU module as a C++ class derived from `Cpu` and override several virtual methods. See Sec. 6.5 for details.

5.3.5 Timing

Modeling time delays is different in firmware and hardware modules. In hardware modules just use a `wait(`*duration*`)` call. There are the `waitUntil()` and `waitOrUntil()` calls which allow to wait for certain events.[1] In firmware modules this is not allowed, because it would stand for a wondrous jump in time.

How is a delay programmed in software? One way is a simple delay loop. This is easily modeled with a call to `requireCpu()`. Note that such a delay must not be modeled as an actual loop in your C++ code since it would only load your simulation host CPU but doesn't advance the simulation time.

More complicated delay loops may have a dedicated break condition to wait for a certain event (i.e., a flag set by an ISR or a dedicated timer module). This can be implemented with the `requireCpuUntil()` and `requireCpuOrUntil()` methods.

Even more sophisticated delays utilize low power modes of the CPU. This sets the CPU to a low power mode which halts the execution until an interrupt occurs. This will wake up the CPU which then continues its operation.

Note that time only elapses in `wait()` and `requireCpu()` calls. Every C++ code you place between such calls virtually runs in the same time instant, i.e. no time elapses.

[1] These `wait()` calls are exit points of the coroutine.

5.3.6 Interrupts

The interrupt handling is implemented in several steps. Within the modeled microcontroller numerous *interrupt sources* exist. These come from some other modules (e.g. a timer, an analog-digital-converter, ...). They are mapped to *interrupt vectors*. Every vector has a *priority* and an *interrupt service routine* (ISR) assigned. Everything except the registration of the ISRs is realized within the CPU module (as described above).

Every (software) module can register its ISR for an interrupt vector. When the interrupt source triggers an interrupt request, the CPU module maps it to the appropriate interrupt vector, checks its priority and then transfers control to the ISR. The currently running CPU task (read: the delay and power consumption) is interrupted and finished later.

5.3.7 Environment

All nodes are placed at 3D positions within the *Environment*. This manages the outer world of all nodes including their surrounding and the RF channel.

Besides the nodes themselves also other objects like walls, floors, trees, interferers, heaters, light sources, ... reside within the environment. Additionally global properties (e.g. the attenuation exponent b (see Sec. 5.3.8)) are defined. Everything is setup and configured with the scripting interface.

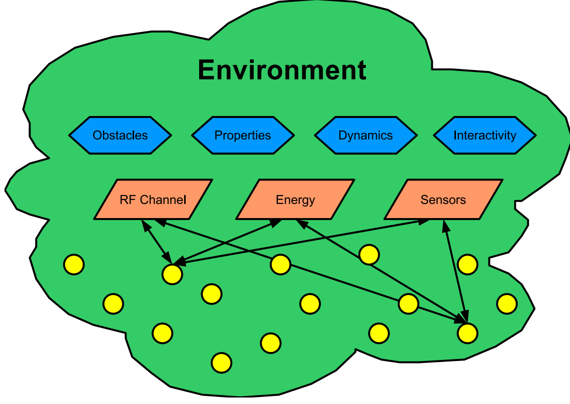

Figure 5: The Environment with functions, objects and sensor nodes

5.3.8 Air

The *Air* is a subcomponent of the environment to handle the RF channels, which are defined by the node placement and the obstacles between them. The RF signal is subject to wave propagation phenomenons like attenuation, reflection, refraction, scattering and fading (multi-path propagation) from the transmitter to the receiver.

Theory The current implementation of the Air only handles attenuation effects, because the other effects would require an enormous complexity in the simulator. For free space propagation the receipient power is defined by

$$P_{\mathrm{Rx}} = P_{\mathrm{Tx}} G_{\mathrm{Tx}} \left(\overrightarrow{\varphi_{\mathrm{Tx,Rx}}}\right) \cdot G_{\mathrm{Rx}} \left(\overrightarrow{\varphi_{\mathrm{Rx,Tx}}}\right) \left(\frac{\lambda}{4\pi}\right)^2 d^{-b} \quad (1)$$

where $G_{\mathrm{Tx}} \left(\overrightarrow{\varphi_{\mathrm{Tx,Rx}}}\right)$ is the antenna gain of the transmitter in the direction to the receiver and $G_{\mathrm{Rx}} \left(\overrightarrow{\varphi_{\mathrm{Rx,Tx}}}\right)$ is the antenna gain of the receiver in the direction to the transmitter. λ is the wavelength, d is the distance between the two nodes and b is the attenuation exponent. The latter is usually 2 for ideal free space propagation, but for indoor environments higher values like 3.5 are more appropriate.

Alternatively the received power can be calculated by dividing the transmit power by the surface of a sphere $4\pi d^2$ (replacing the exponent by b) with radius equaling the distance of the receiver and transmitter d. This "diluted" power density is then multiplied by the antenna area A_{Rx} of the receiver.

$$P_{\mathrm{Rx}} = P_{\mathrm{Tx}} \frac{1}{4\pi} d^{-b} A_{Rx} \quad (2)$$

The attenuation from the transmitter j to receiver i is given by

$$A_{i,j} = \frac{P_{\mathrm{Rx},i}}{P_{\mathrm{Tx},j}} \quad (3)$$

where i and j are the index of the node ($i, j \in 1, 2, \ldots, n$). Calculating the attenuation from any node to every other node gives the *adjacency matrix*

$$\underline{A} = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \cdots & A_{n,n} \end{pmatrix} \quad (4)$$

Let the vectors $\overrightarrow{P_{\mathrm{Tx}}}$ and $\overrightarrow{P_{\mathrm{Rx}}}$ hold the transmitted and received power of all n nodes, respectively. Then the calculation of received power is easily written by the matrix operation

$$\overrightarrow{P_{\mathrm{Rx}}} = \underline{A}\, \overrightarrow{P_{\mathrm{Tx}}} \quad (5)$$

Since the results only make sense when a single transmitter is active, only one value in $\overrightarrow{P_{\mathrm{Tx}}}$ is different from zero and the matrix multiplication in (5) reduces to a single column scaling operation.

The adjacency matrix is a precisely defined interface from the environment to the data communication. For an actual environment setup $\underline{A}$ is calculated. This is used during the network simulation as shown in (5). Due to this simple interface, $\underline{A}$ can also be calculated by an external RF channel simulation tool.

The current implementation of the Air only supports isotropic antennas with uniform antenna gain. No obstacles are considered for the adjacency matrix, though explicitly given additional attenuations between every pair of two nodes are supported.

Every receiving node is notified of a data packet transmitted via the Air. This notification contains the receiver power. The node then calculates the signal to noise ratio (SNR) with its own and the received noise power. From this SNR the bit error ratio (BER) is derived. The BER is a function of the SNR which depends on the modulation format. From the BER the bit error count is calculated.

Implementation The implementation is described in Sec. 6.10.

5.3.9 Power Simulation

Every task simulates its power consumption. At simulation startup it registers at its power supply module. This enables a hierarchy of power distribution. During simulation the actual consumption is calculated (by your C++ code) and reported to the power supply. This propagates up the supply tree and is finally reported to the central *Power Meter*.

The consumed power equals $P = U \cdot I$. The supply voltage U is provided by the supply module. Every consumer can have different electrical

behavior. i.e. the current I depends in different ways on the supply voltage. Currently three kinds of *power reporters* are provided but you are free to define new ones.

- `ConstantReporter`: $I = I_{\text{const}}$. The current does not depend on the supply voltage.
- `ResistiveReporter`: $I = \frac{U}{R}$. The supply current is proportional to the supply voltage, typically as for a resistor.
- `LinearReporter`: $I = I_{\text{const}} + \frac{U}{R}$. A constant current plus a supply voltage dependant portion.
- Define your own reporter by deriving from the `PowerReporter` base class. This can implement any non-linear current characteristics you need.

The power supply module calculates its output voltage depending on the output current, therefore different source behaviors are possible too. This results in a simple electrical network.

At any point in a hardware task you can set a new power reporter and its characteristics. The previous reporter is then replaced by a new one. The reporter handles all supply voltage changes according to its properties automatically.

Note that only tasks which simulate dedicated hardware consume power on their own. Tasks which simulate software tasks just use the `requireCPU()` method. The CPU module will report its power consumption and the additional info for which task it is executing code.

6 Example

In this section we will discuss the implementation of a simple application doing some processing in a CPU and blinking a LED in parallel. This will demonstrate all important concepts of the PAWiS Framework. For the description of the Environment and the Air another example is introduced.

6.1 Modules

As shown in Fig. 6 the system is built of the following modules: The main program is running in the Application module ("App") which is a pure software module. This is connected to the Timer and the LED to invoke their functions via their functional interfaces. All modules (except the App) are supplied by the power source "LDO" (low drop-out regulator). The timer can issue an interrupt which interrupts the application program and starts the ISR (interrupt service routine).

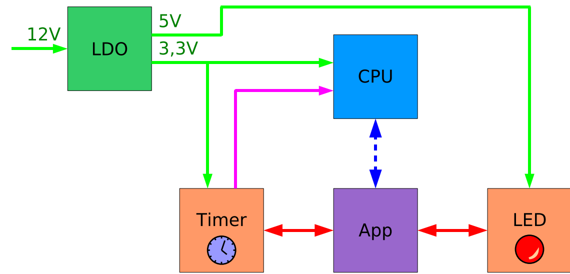

Figure 6: Modules of the LED example.

Every module is implemented as a C++ class derived from the PAWiS Framework class `PawisModule` (see Lst. 1). Inside of the class you have to use a special macro `Module_Class_Members(`*class*`,` *parent*`, 0)` which declares the constructor and some other internal stuff. Its first parameter is always the name of the class, the second parameter is the parent class and the third must be `0`.

In the class definition (see Lst. 3) you have to add the macro `Define_Module(`*class*`)`.

The class can contain as many member variables and methods as you like.

Listing 1: Class declaration of the PAWiS module "Led" from `led.h`.

```
#include <memory>
#include <base/paModule.h>
#include "ledReporter.h"

using namespace pawis;

class Led : public PawisModule {
  private:
    bool m_bState;
    std::auto_ptr<PowerSourceAdapter>
      m_pPowerAdapter;

    void set(TaskControl &pa_oControl);

    virtual void onStartup();
    virtual void onInit();

  public:
    Module_Class_Members(Led, PawisModule, 0)
};
```

There are two virtual methods which you can override. They are automatically called upon start of the simulation. `onStartup()` is first executed for all modules, and then `onInit()` is executed for all modules. Therefore in `onStartup()` you have to register all your interfaces, power sources, ... offered to other modules. In `onInit()` you can refer to the previously registered items (of other modules). Read on for some examples.

6.1.1 The NED File

The module instantiation and connections between each other are described in the so called *NED file*. This is a text file used by the underlying OMNeT++ framework to setup the whole network.

A NED file defines three types of objects which are setup within a hierarchy. The highest level is the *network* which we use to connect the individual nodes. It is declared in a block of `network` *name* `:` *top module* `endnetwork` (see Lst. 2). See also Sec. 5.3.7 how to build complex networks.

The next lower level are *compound modules* declared with `module` *name* ... `endmodule` blocks. They combine other compound or simple modules and set their parameters and interconnect.

Simple modules are the lowest level in the NED hierarchy. These are the only items which actually implement functionality. Therefore they are the only items which have individual C++ code provided by the programmer. That means that you will concentrate on the implementation of simple modules. To structure your network nodes you define compound modules in the NED file.

The NED file is translated with the `nedtool` tool (provided by OMNeT++) which creates a C++ source file with the postfix `_n.cpp`. This contains auto-generated C++ class declarations for the described compound modules and networks. It must be compiled the same way as your other C++ source files.

A network node is built as a compound module (see `LedNode` in Lst. 2). You can define one or more different types of network nodes. The compound module representing the network (here `LedNetwork`) can instanciate any number of these nodes. Simply ensure that your module code implements interoperability between the different kinds of nodes.

Listing 2: NED file for the LED example from `Led.ned`.

```
simple SimpleCpu
endsimple

simple Led
  gates:
    in:  App_in;
    out: App_out;
endsimple

simple App
  gates:
    in:  Timer_in;
    in:  Led_in;
    out: Timer_out;
    out: Led_out;
endsimple

simple Timer
  gates:
    in:  App_in;
    out: App_out;
endsimple

simple Ldo
endsimple

simple Battery
endsimple

simple Config
  parameters:
    ConfigFile: string;
endsimple

module LedNode
  parameters:
    NodeId : const;
  submodules:
    cpu     : SimpleCpu;
    led     : Led;
    app     : App;
    timer   : Timer;
    ldo     : Ldo;
    battery : Battery;
  connections:
    app.Led_out   --> led.App_in;
    app.Led_in    <-- led.App_out;
    app.Timer_out --> timer.App_in;
    app.Timer_in  <-- timer.App_out;
endmodule

module LedNetwork
  submodules:
    config : Config
      parameters:
        ConfigFile = "";
    ln1: LedNode
      parameters:
        NodeId = 1;
endmodule

network lednet : LedNetwork
endnetwork
```

6.1.2 Messages

Modules can commmunicate each other by sending messages (through the corresponding gates). Each abstraction layer requires a message class, which is derived from the Pawis Framework `BaseMessage` class. Each implementation includes the specific header fields for each layer. Messages can be easily implemented by writing a simple `.msg` file which is translated by the `opp_msgc` OMNeT++ application into a C++ file with autogenerated methods to access the header fields.

6.2 Tasks

Every module is built of *tasks*. Each of these tasks is implemented as a method inside of the module's class. Their name can be chosen freely but they must accept exactly one parameter of the type `TaskControl&`. The return type is `void` (see `App::myMain()` and `App::onTimer()` in Lst. 3).

The `TaskControl&` parameter holds an object which is a unique representation of the task. It contains the status and context of the task as well as some task-related methods like `invoke()` (see below).

Tasks are virtually running in parallel as described in Sec. 5.3.2. Their execution leaves at certain (defined) points ("yield"). The simulation kernel then hands over control to other tasks. The execution of the task is then continued where it was interrupted.

Points where the execution is transferred to other tasks are

- `wait(`*delay*`)`
- `waitUntil(`*predicate*`)`
- `waitOrUntil(`*delay*, *predicate*`)`
- `requireCpu()`
- `requireCpuUntil()`
- `requireCpuOrUntil()`
- `invoke()` and `invokeReturn()` with various combinations of parameter types (see Sec. 6.3.2).

Please see the Doxygen documentation of `TaskControl` for details and the exact parameter lists of these methods.

Within every task you can have loops, `if` and `switch` conditions, ... Note that you must include yield points as described above or return from the method after performing the desired job. Otherwise your C++ method will never give back control to the simulation kernel (this is why we call this cooperative multitasking). When the method exits (e.g. with `return`), the task is finished.

Every task models either a *dedicated hardware* block (e.g. a timer, an AD converter, ...) or some *software* running on the CPU (e.g. the routing layer of your protocol). Within one module you can have as many tasks and mix hardware and software tasks as you need.

All tasks can access the member variables of the module class instance which are shared amongst the tasks. This is even possible when the same task method is running multiple times. For variables which are unique to every task instance use the stack, i.e. declare local variables within the method. Note that every task has its own stack, which is limited. To set the stack size, the functions `startTask()`, `registerIsr()` and `registerFunctionalInterface()` offer an additional parameter. Its default value currently is 32kB.

A task can be started in two different ways. Tasks that should "just run" from the beginning on are started from the `onInit()` method (see Lst. 3) using the `startTask(`*name*`,` *task method*`,` *params in*`,` *params out*`)` call. Note that this function can be used anywhere and anytime to create new tasks. Its first parameter is the method (using a macro to cast to the proper type). The second and third parameters are pointers to `ParameterList` objects for input and output parameters, respectively. They can be "`0`" and are discussed in detail with functional interfaces.

The second kind of task creation is using functional interfaces and is described below.

A call to `startTask()` is always non-blocking which means that the started task runs in parallel and independently. The starter does not wait for any condition (e.g. finishing) of it.

Listing 3: Implementation of the PAWiS module "App" from `app.cpp` showing a task.

```
#include "app.h"

Define_Module(App);

void App::onInit() {
  // init member variables
  // ...
  // register ISRs
  registerIsr(CAST_TASK(App::onTimer), 17,
    "timer_isr");
  // schedule initial task
  startTask("main",CAST_TASK(App::myMain),0,0);
}

void App::onTimer(TaskControl &pa_oControl) {
  // do our internal work
  m_bLedState = !m_bLedState;

  // set the LED
  pa_oControl.requireCpu(0.5, 0.0, 0.2, 0.3,
    0.001f);
  pa_oControl.invoke("Led", "set", m_bLedState);

  // setup new timer event
  pa_oControl.requireCpu(0.5, 0.0, 0.3, 0.2,
    0.005f);
  pa_oControl.invoke("Timer", "start",
   (unsigned int)(20));

  ++m_nCounter;
}

// predicate function for requireCpuOrUntil()
bool App::done() {
  // test an external condition,
```

```
  // e.g. ADC has finished conversion
  if (m_nCounter > 2) {
    return true;
  }
  return false;
}

void App::myMain(TaskControl &pa_oControl) {
  // ...
  // start the LED toggling
  onTimer(pa_oControl);
  // Before Start waiting 50ms or the predicate
  pa_oControl.requireCpuOrUntil(0.1f,0.0f,0.0f,
    0.9f, 0.05, CAST_PREDICATE(App::done));
  // main program running infinitely
  while (true) {
    pa_oControl.setCpuState(
      new SimpleCpu::CpuState(
        SimpleCpu::CpuState::pmSleep));
    pa_oControl.requireCpu(/* ... */, 1.0e-6);
  }
}
```

6.3 Functional Interfaces

The Timer as well as the LED offer a functional interface each which is used by the Application (red arrows in Fig. 6, method `set()` in Lst. 1 and `startTimer()` in Lst. 4).

6.3.1 Setup

The functional interface must be registered in `onStartup()` (and not in `onInit()`) with `registerFunctionalInterface(`*name*`,` *method*`,` *multiInvokable*`)` (see Lst. 4). The first parameter is a string with the name of the interface (referred to by other modules). The second parameter is a casted pointer to the task method. The third parameter tells the framework, if this functional interface is intended to be invoked multiple times in parallel. When set to `false` the framework does additional sanity checks to warn the user of multiple invocations.

6.3.2 Invocation

Other modules can then invoke the functional interface with the method `invoke(`*module*`,` *interface*`,` *params in*`,` *params out*`)` (see `onTimer()` in Lst. 3). There are multiple overloaded versions of `invoke()` for several combinations of parameters as well as `invokeReturn<typename>()` methods for typed return values. This are convenience functions that you don't have to setup a parameter list. For details please consult the Doxygen documentation of `TaskControl`.

Whenever a functional interface is invoked from another module, a new task is created and its method (here `Timer::startTimer()`) is executed. When this method returns, `invoke()` returns too and execution of the caller continues. That means that `invoke()` is always blocking (see both calls of `invoke()`).

6.3.3 Parameters

The task `App::onTimer()` also shows how to supply parameters to a functional interface. The first functional interface invocation to `Led.set` supplies one parameter of type `bool`. Here one of the above mentioned overloaded versions of `invoke()` is used.

Alternatively a `ParameterList` object could be created and the parameter(s) added with the `add<`*type*`>(`*value*`)` template method. When the invocation returns the input parameter list is consumed, i.e. the added parameters are cleared. This means that the ownership of the parameters is transferred from the caller to the callee, which is responsible for `delete`'ing reserved memory. This is not relevant for the overloaded `invoke()` methods with templated parameter types. The `ParameterList` object can be reused for the next invocation of another functional interface. For convenience `Timer.start`, which requires an `unsigned int` specifying the delay in milliseconds (see Lst. 4) is again called with an overloaded `invoke()`.

Listing 4: Implementation of the PAWiS module "Timer" from `timer.cpp`.

```
#include "timer.h"

/* ... */

void Timer::onStartup() {
  // register the functional interface
  registerFunctionalInterface("start",
    CAST_TASK(Timer::startTimer), false);
}

void Timer::onInit() {
  // setup interrupt source
  m_oTimerSource.setName("timer");
  // register our interrupt source
  registerIntSource(m_oTimerSource);
  // Create power source adapter and reporter
  m_pPowerAdapter.reset(new PowerSourceAdapter(
    *getContext().findPowerSource("ldo"),
    LdoPowerSource::src3V3, 0, true));
  m_pPowerAdapter->setReporter(
    new LinearReporter(*this));
  // Initial consumption
  m_pPowerAdapter->
    getReporter<LinearReporter>()->
      set(0.0f, /* infinite */);
}

void Timer::countTimer(TaskControl &pa_oControl) {
  // start the timer
```

```
  m_pPowerAdapter->getReporter<LinearReporter>().
    set(2.2e-3, 1.5e6);
  // show that we are running
  setFillColor("#FF0000");
  // wait for the delay
  pa_oControl.wait(m_tDelay);
  // timer has run off
  m_pPowerAdapter->getReporter<LinearReporter>().
    set(1.2e-3, 1.5e6);
  // show that the timer ran off
  setFillColor("#00FF00");
  // issue an interrupt request
  pa_oControl.intRequest(m_oTimerSource);
}

void Timer::startTimer(TaskControl &pa_oControl) {
  // get the delay parameter and calc the seconds
  m_tDelay = pa_oControl.inParams()->
    get<unsigned int>(0) / 1000.0f;
  // start parallel task doing the delay
  startTask(CAST_TASK(Timer::countTimer), 0, 0);
}
```

The implementation of a functional interface is shown in Lst. 4. The (previously registered) method `Timer::startTimer()` is executed upon the invocation of the functional interface `Timer.set`. This extracts the first parameter (0) from the parameter list held by the `TaskControl&` object and stores it to a member variable. It is important that the type (supplied as a template parameter) must exactly match to the one supplied to the according `invoke()`.

Secondly a new task is created with `startTask()`. It will run in parallel (and therefore independently) to the functional interface task which allows the method to return immediately.

6.4 Timing

6.4.1 Hardware tasks

As stated in Sec. 5.3.5 hardware tasks model time delays with the methods `wait(`*delay*`)`, `waitUntil(`*predicate function*`)` and `waitOrUntil(`*delay*`,`*predicate function*`)`. The first function delays execution for a constant amount of time, the second function delays execution until a certain condition (tested by the predicate function) is fulfilled. The third form returns after a constant delay or until the condition is true, whichever happens first. In the task method `Timer::countTimer()` in Lst. 4 the timer delay is realized with the `wait()` method.

The parameter of `wait(Or)Until()` (and `requireCpu(Or)Until()` is a so called *predicate function* which tells the simulation kernel whether the (modeled) job has finished or not (see `App::done()`). This is evaluated as long as `wait(Or)Until()` is "active". The function has to return `true` to stop and `false` to continue `wait(Or)Until()`.

Usually the predicate function has to reside within the same module where it is used. With a derived class from `TaskControl::Predicate` (e.g. the available `TaskControl::ModulePredicate`) you can implement predicate function across module boundaries.

Note that both, `wait()` and `wait(Or)Until()`, are blocking functions which means that simulation time elapses whenever they are used. Therefore they are yield points within the task method.

6.4.2 Software tasks

In software tasks a delay is modeled differently as already described in Sec. 5.3.5.

6.4.3 Trigger

As stated in Sec. 5.3.3 one task can wait for a trigger condition satisfied by another task. This is modeled by two separate tasks. The first task is started from `onInit()` and uses a `waitUntil()` to wait for the trigger. The second task implements a functional interface and sets a class member variable which is queried by the predicate function used with the `waitUntil()`.

6.5 CPU

As described in Sec. 5.3.4 the CPU usage is simulated with the `requireCpu(`*percentage integer*`,` *percentage float*`,` *percentage memory access*`,` *percentage flow control*`,` *duration*`)` method from the `TaskControl&` object. In the method `App::onTimer()` in Lst. 3 this call is used twice. At first the software effort to set the LED's state is modeled, and the second call models the setup of the timer.

`requireCpu()` is useful for tasks where the total execution time is principally determinable before its start. For CPU tasks which run until a certain external condition set their end, the `requireCpuUntil(`*percentage integer*`,` *percentage float*`,` *percentage memory access*`,` *percentage flow control*`,` *predicate function*`)` method is used, similar to the `waitUntil()` function.

As already mentioned there is also a combined delay and condition function `requireCpuOrUntil()` (see Lst. 3, `App::myMain()`). The example shows how to model a task which is waiting for an external condition (e.g. a counter value is reached, the PHY has finished transmission, ...). To avoid infinite hangs a certain timeout is used as second exit condition. Lst. 5 shows how such firmware code could look like. `P1IN` is a digital 8-bit input port where the bit nr. 5 will go high when the

external peripheral has finished its job. The variable `timeout` is used to count upwards and is compared with the constant `TIMEOUT_50ms`. Its value is precalculated so that the loop lasts for 50ms when executed `TIMEOUT_50ms` times.

Listing 5: Sample code showing a waiting loop with external condition and timeout.

```
timeout = 0;
while (((P1IN & 0x20) == 0) &&
       (timeout < TIMEOUT_50ms)) {
  timeout++;
}
```

Note that the `requireCpu()` functions are blocking, which means that simulation time elapses during their execution. Therefore they are yield point of a task.

The numbers given to `requireCpu()` are used by the CPU implementation to determine the real execution time and power consumption. The first four numbers reflect the proportions of different processing types. You have to estimate them depending on your real code. Ensure that they sum up to 1.0.

The fifth parameter gives the total execution time referred to the norm CPU. To determine this you have several options. The most accurate is surely using a CPU simulation tool and measure the time for a certain task programmed as firmware code. Alternatively you can use real hardware and measure the processing time with an oscilloscope.[2]

Since the contribution of the CPU to power consumption and timing is rather small, these values don't have to be very accurate. Therefore you can also estimate the durations yourself.

6.5.1 Callbacks

A *callback* is a method which is called after a certain amount of time. It is scheduled with `scheduleCallback(`*method*`,`*delay*`)` and can be canceled with `cancelCallback()`. For further details consult the Doxygen documentation of `PawisBase`.

6.5.2 CPU Module Implementation

The CPU implementation is a class derived from the PAWiS base class `Cpu`. You have to implement certain virtual methods to customize the simulation model. For a very simple CPU you only have to implement `onInit()`, `calcCpuTime()` and `getIntVectorPriority()`.

[2]Set a digital output of your micro controller high directly before your routine and back to low directly afterwards. Attach the oscilloscope to this output pin and use its measurement tools to get the duration of this "pulse".

The method `calcCpuTime()` has to calculate the real execution time of a processing request depending on the desired CPU characteristics. In our example (see Lst. 6) we implement a CPU with the same performance as the norm CPU and therefore simply use the norm-CPU referred execution time. Please consider the Doxygen documentation of `FuncCpuMessage` on how to retrieve the percentage values of the individual processing domains.

Since the CPU can handle interrupts, the prioritizing of concurrent interrupts requires classification of the individual interrupt vectors. You have to implement this in `getIntVectorPriority()`. Again, we stay simple and prioritize the vectors according to their numerical value.

The initialization of the module in `onInit()` has to assign an interrupt mapping as described in Sec. 6.7. This can also be done from another module by retrieving a pointer to the CPU instance. Here we also setup a power reporter, fully analogous to Sec. 6.6.1.

A special feature of our simple CPU is the sleep state. The user of the CPU uses `TaskControl::setCpuState(TaskControl::CpuState)` to set a new state (see Lst. 3). Note that the parameter is a class to allow unlimited features. We derive from this class and provide two different modes `pmActive` and `pmSleep`. Every time the user uses `setCpuState()` the method `onStateChange()` (see Lst. 6) is executed. Depending on the supplied mode the according power consumption is reported. Finally the CPU execution is either `pause()`d or `resume()`d.

To reactivate the CPU in case of an interrupt the method `onIsrEnter()` uses the internal method `Cpu::setCpuState()`. This implementation results in the behavior that the CPU is woken by an interrupt and then stays active, even if the ISR has finished. Therefore in `App::myMain()` the power mode is set again in an infinite loop (see Lst. 3). The method also shows an important fact: You have to use `requireCpu()` after deactivating the CPU to simulate the stopped execution. That means that simply calling `setCpuState()` only internally stores that the CPU is deactivated. To experience this fact a processing request has to be performed which will not start until the CPU is reactivated again.

You could also override the method `onIsrLeave()` and set the power mode to `pmSleep` again. Note that in this case the main program could not wake up again, because every interrupt deactivates the CPU after it has finished. To solve this problem the ISR can decide whether the CPU should stay active after its end or shut-down again.

To implement this you have to utilize the full power of using a class `CpuState` for `setCpuState()`. With this you can communicate the desired behavior to the `Cpu` module. In Lst. 6 an immediate reaction of the ordered power mode is accomplished in `onStateChange()` whereas in the above case the desired behavior of a future event has to be stored and then handled in `onIsrLeave()`. The current power mode as well as the power mode after returning from the ISR have to be stored in a state stack which is manipulated using a special `CpuState` class. An example of this topic is the `CpuSimple` of the Module Library.

Listing 6: Declaration and implementation of the Cpu module from `cpu.h`.

```
/* ... */
#include <base/paCpu.h>
/* ... */

class SimpleCpu : public Cpu {
  public:
    class CpuState
      : public TaskControl::CpuState {
      public:
        typedef enum {
          pmActive,
          pmSleep
        } t_PowerMode;
      private:
        t_PowerMode m_PowerMode;
      public:
        CpuState(t_PowerMode pa_PowerMode) {
          m_PowerMode = pa_PowerMode;
        }
        t_PowerMode getPowerMode() const {
          return m_PowerMode;
        }
    };
  private:
    std::auto_ptr<PowerSourceAdapter>
      m_pPowerAdapter;
    InterruptMapping m_oIntMapping;
  protected:
    virtual t_Time calcCpuTime(
      const FuncCpuMessage &pa_msgCpu)
    {
      return pa_msgCpu.getDuration();
    }

    virtual int getIntVectorPriority(
      TaskControl::t_IntVector pa_intVector)
      const
    {
      return pa_intVector;
    }

    virtual void onStateChange(/* ... */) {
      const CpuState* stateNew =
        dynamic_cast<const CpuState*>
        (pa_stateNew);
      if (stateNew->getPowerMode() ==
          CpuState::pmActive) {
        // CPU is active
        m_pPowerAdapter->
          getReporter<LinearReporter>()->
            set(3.2e-3, 1.5e3);
        resume();
      } else {
        // CPU in sleep state
        m_pPowerAdapter->
          getReporter<LinearReporter>()->
            set(5e-5, 1e6);
        pause();
      }
    }

    void onIsrEnter() {
      setCpuState(new CpuState(CpuState::pmActive));
    }

    virtual void onInit() {
      // setup power reporter
      m_pPowerAdapter.reset(
        new PowerSourceAdapter(
          *getContext().findPowerSource("ldo"),
          LdoPowerSource::src3V3, 0, true));
      m_pPowerAdapter->setReporter(
        new LinearReporter(*this));
      // setup interrupt mapping
      m_oIntMapping.setMapping("timer", 17);
      setInterruptMapping(&m_oIntMapping);
      // set initial state
      setCpuState(
        new CpuState(CpuState::pmActive));
    }

  public:
    Module_Class_Members(SimpleCpu, Cpu, 0)
};
/* ... */
```

6.6 Power Simulation

6.6.1 Power Consumption

Only hardware tasks draw power on their own. The power consumption of software tasks is modeled by the CPU module which is activated with the `requireCpu()` calls.

Permanent hardware tasks have to register at their power supply with a call to `TaskControl::subscribePowerSource(`*source name*, *source number*`)`. The source name is a string which selects the supply module. Every source can implement several outputs (e.g. an LDO with two different output voltages sourcing from one input). The second parameter is an integer specifying the supply module's output number.

Afterwards the consumption reporter is set and configured. The template method

`TaskControl::getReporter<type>()` returns a reference to the internally stored reporter object where *type* is the reporter class type (see Sec. 5.3.9). If you use different types within one task after another the previous reporter object is `delete`d and a new one is created. The reference is used to execute the reporter's `set(...)` method which accepts the parameters according to the reporter's properties.

Reporting the consumption means that you specify "from now on I'm consuming n mA current". This is just the start of a periode. Its end is determined only by another `set()` invocation in the same task.

Note that with the destruction of the `TaskControl` object (which happens when your task has finished), the `PowerReporter` object is also destroyed. That means that the power consumption of that task also stops. Therefore the described method is only applicable for tasks which are started once and will run forever.

On the other hand you might as well implement tasks which only run for a certain while, e.g., to start some timer (see Lst. 4) or to switch a device on or off (see Lst. 7). Such tasks are usually invoked externally. In that case the task exits while the current consumption should continue. To implement this behavior you have to instantiate your own `PowerReporter` and `PowerSourceAdapter` objects.

Lets have a look at Lst. 7. The LED current consumption is updated in the `set()` functional interface. If it is switched on, the forward voltage U_f is set to 2.2V and the series resistor to 280Ω. To switch off the LED its series resistor is set to ∞. This is a trick because we don't want to simulate a switch. Note that the method `set()` immediately exits and assumes that the power consumption continues.

Therefore in the method `onInit()` a `PowerSourceAdapter` object is created. This is necessary to connect a `PowerReporter` to a `PowerSource`. This `PowerReporter` is then assigned to the `PowerSourceAdapter` using `setReporter()`. A fully analogous case can be found in Lst. 4 where the method `countTimer()` implements a waiting timer task. It is started by `startTimer()` and sets the power consumption of the *running* timer. Then it uses `wait()` to implement the timer delay and finally reports the power consumption of the *inactive* timer, issues an interrupt request end exits. This inactive power consumption has to continue until it is next invoked.

Listing 7: Class implementation of the PAWiS module "Led" from `led.cpp`.

```
/* ... */

void Led::onStartup() {
  registerFunctionalInterface("set",
    CAST_TASK(Led::set), false);
}

void Led::onInit() {
  // Create power source adapter and reporter
  m_pPowerAdapter.reset(
    new PowerSourceAdapter(
      *getContext().findPowerSource("ldo"),
      LdoPowerSource::src5V, 0, true));
  m_pPowerAdapter->setReporter(
    new LedReporter(*this));
  // Initial consumption
  m_pPowerAdapter->
    getReporter<LedReporter>()->
      set(0.0f, /* infinity */);
}

void Led::set(TaskControl &pa_oControl) {
  m_bState = pa_oControl.inParams()->
    get<bool>(0);
  if (m_bState) {
    m_pPowerAdapter->
      getReporter<LedReporter>()->
        set(2.2f, 280.0f);
    setFillColor("#40ff40");
  } else {
    m_pPowerAdapter->
      getReporter<LedReporter>()->
        set(0.0f, /* infinity */);
    setFillColor("#206020");
  }
}
```

6.6.2 Power Sources

In your simulation most power sources will be converters (LDO, DC/DC), i.e. no real sources. But you will have at least one real power source (like a battery or a solar cell). To build such *power sources* a class derived from `PowerSource` or `PendingPowerSource` is implemented (see Lst. 8). These base classes are used for real sources (e.g. batteries) and for sources which are themselves consumers (e.g. LDOs or buck converters), respectively.

In our example the `LdoPowerSource` implements two sources (`NumSources`) with 5V (`src5V`) and 3.3V (`src3V3`). These constants are used by the consumers (see Lst. 4) and in `calcVoltage()` and `calcCurrent()` (see Lst. 9).

The `LdoPowerSource` is itself a consumer and therefore requires its own power source (see its constructor). `BatterySource` is the origin of energy and thus doesn't have another source.

The power source classes are instantiated in the normal `PawisModule Supply` (see also Lst. 9).

Listing 8: Declaration of the power source module "Ldo" from `ldo.h`.

```
/* ... */
#include <base/paPowerSource.h>
#include <base/paLinearReporter.h>
#include <base/paPendingPowerSource.h>
#include <base/paLimitCycleBehaviour.h>
/* ... */

class LdoPowerSource : public PendingPowerSource {
  public:
    static const unsigned int NumSources = 2;
    static const unsigned int src5V      = 0;
    static const unsigned int src3V3     = 1;
  private:
    float m_fSourceImpedance5V;
    float m_fSourceImpedance3V3;

  protected:
    virtual t_Voltage calcVoltage(/* ... */);
    virtual t_Current calcCurrent(/* ... */);

  public:
    LdoPowerSource(
      PawisModule& pa_oModule,
      PowerSource* pa_pPowerSource,
      unsigned int pa_iPowerSourceNum);
};

class BatterySource : public PowerSource {
  protected:
    LinearReporter m_oReporter;
    virtual t_Voltage calcVoltage(/* ... */);
  public:
    BatterySource(PawisModule& pa_oModule);
};

class Supply : public PawisModule {
  private:
    BatterySource*  m_pBatterySource;
    LdoPowerSource* m_pLdoSource;
    void onStartup();
  public:
    Module_Class_Members(Supply, PawisModule, 0)
};

/* ... */
```

Every supply class has to implement the method `calcVoltage()` which is called whenever an output current changes. In this example we implement two simple sources with constant output resistance `m_fSourceImpedance5V` and `m_fSourceImpedance3V3` (see Lst. 9). Since the `LdoPowerSource` itself is a consumer too, the method `calcCurrent()` also has to be implemented. This must return the total current consumption. Its parameter `OutputSource` carries the information which output has changed so that the input current must be recalculated. In the example we have stored the individual output currents in member variables and simply return the sum of them plus a constant quiescent current.

The `BatterySource` is implemented as a never ending 12V source. The reporting of the energy origin is done with `m_oReporter`). Note that the reporter is instantiated without a source, therefore we have to set the input voltage by hand with `setVoltage()`.

In `LdoPowerSource`'s constructor we have to assign an *update behavior* to our source. As described in Sec. 5.3.9 the output voltage of a source can change on the current load. The current load itself also depends on the output voltage, therefore iterative calculation of the true values is performed. The update behavior class determines how this updates are performed. Here we use a class which limits the count of iterative cycles to 2.

Listing 9: Implementation of the power source module "Ldo" from `ldo.cpp`.

```
/* ... */

/**** Ldo Module ****************************/

Define_Module(Supply);

void Supply::onStartup() {
  // create and register the battery source
  m_pBatterySource = new BatterySource(*this);
  m_oThisContext.registerPowerSource("battery",
    *m_pBatterySource);
  // create and register the LDO source
  m_pLdoSource = new LdoPowerSource(*this,
    m_pBatterySource, 0);
  m_oThisContext.registerPowerSource("ldo",
    *m_pLdoSource);
}

/**** LdoPowerSource ************************/

t_Voltage LdoPowerSource::calcVoltage(
  unsigned int pa_nSource,
  t_Current pa_fCurrentSum)
{
  t_Voltage ret;
  switch(pa_nSource) {
    case src5V:
      ret = 5.0f -
        pa_fCurrentSum * m_fSourceImpedance5V;
      break;
    case src3V3:
      ret = 3.3f -
        pa_fCurrentSum * m_fSourceImpedance3V3;
      break;
    default:
      // error
  }
  return ret;
}

t_Current LdoPowerSource::calcCurrent(
  unsigned int pa_nOutputSource)
{
  return (calcCurrentSum(src5V) +
```

```
        calcCurrentSum(src3V3) +
        m_fSourceCurrent3V3 + 10e-6);
}

LdoPowerSource::LdoPowerSource(
  PawisModule& pa_oModule,
  PowerSource* pa_pPowerSource,
  unsigned int pa_iPowerSourceNum) :
  PendingPowerSource(pa_oModule, NumSources)
{
  // Source for our own power consumption.
  setPowerSource(*pa_pPowerSource,
                 pa_iPowerSourceNum);
  // setup internal variables
  m_fSourceImpedance5V  = 1.0f;
  m_fSourceImpedance3V3 = 1.5f;
  // update behavior
  assignBehaviour(new LimitCycleBehaviour(2));
}

/**** BatterySource **************************/

BatterySource::BatterySource(
  PawisModule& pa_oModule) :
  m_oReporter(pa_oModule)
{
  assignBehaviour(new LimitCycleBehaviour(2));
  m_oReporter.setVoltage(12.0);
}

t_Voltage BatterySource::calcVoltage(
  unsigned int pa_nSource,
  t_Current pa_fCurrentSum)
{
  m_oReporter.set(pa_fCurrentSum, 1e6);
  // luckily we have an infinitely full battery
  return m_oReporter.getInputVoltage();
}
```

6.6.3 Power Reporter

As already denoted in Sec. 5.3.9 there are several predefined power reporter classes, e.g. `ConstantReporter`. For special consumption behavior (i.e. non-linear current characteristics) you have to define your own reporter class. The `Led` module shows the usage of the custom `LedReporter` to implement the characteristic of a LED with a series resistor in a simplified way (see Lst. 10).

The reporter is derived from the class `PowerReporter` and has to implement the methods `calcCurrent()` and `set()`. The parameters of `set()` can be defined freely as your simulated consumer requires. Here we consider a (constant) forward voltage of the LED (`pa_fVoltage`) and a series resistor (`pa_fResistorValue`). In `calcCurrent()` the current consumption at a given input voltage (retrieved with `getInputVoltage()`) is calculated. In our case we have to ensure that the current is non-negative.

Listing 10: Declaration and implementation of the custom power reporter "LedReporter" from `ledReporter.cpp`.

```
/* ... */

#include <base/paPowerReporter.h>

class LedReporter : public PowerReporter {
  private:
    t_Voltage m_fVoltage;
    float     m_fResistor;
  public:
    LedReporter(PawisBase &pa_oBase) :
      PowerReporter(pa_oBase),
      m_fVoltage(0.0f),
      m_fResistor(/* infinity */)
    {}

    virtual t_Current calcCurrent() {
      if (getInputVoltage() > m_fVoltage) {
        // valid condition: calc current
        return
          (getInputVoltage() - m_fVoltage)
            / m_fResistor;
      } else {
        // otherwise - no current
        return 0.0f;
      }
    }

    void set(t_Voltage pa_fVoltage,
             float pa_fResistor) {
      m_fVoltage  = pa_fVoltage;
      m_fResistor = pa_fResistor;
      currentChanged();
    }
};
```

6.6.4 Power Logging

Whenever the power consumption of a task changes (i.e., the `set()` method of a power reporter is invoked), it is logged to an external log file together with the time and module and task information. Additionally the requesting module is stored, e.g. when the CPU consumes power on behalf of the routing network layer.

To mark certain points in your processing the `reportEvent(`*string*`)` method can be used. The string is simply stored with a time stamp and module information in the log file.

The logged data is processed and visualized with the DataProcessing tool (see Fig. 7). The graph area shows the power consumption sequence over time. The module hierarchy is shown in the bottom left of the windows. Click with the right button at a modules name to change its color. When you move the mouse in the graph

window, an overlay field displays the current values of all modules at the marked time.

To zoom horizontally or vertically use the shifters at the bottom right of the graph window. Click and hold the right mouse button to pan the image. You can also use the scroll bars for that. The button [Reload] is used to reload the log file (e.g. after another simulation run). Use [Fit Key] to fit the graph horizontally (i.e. time axis) and [FitValue] to fit it vertically (i.e., power axis).

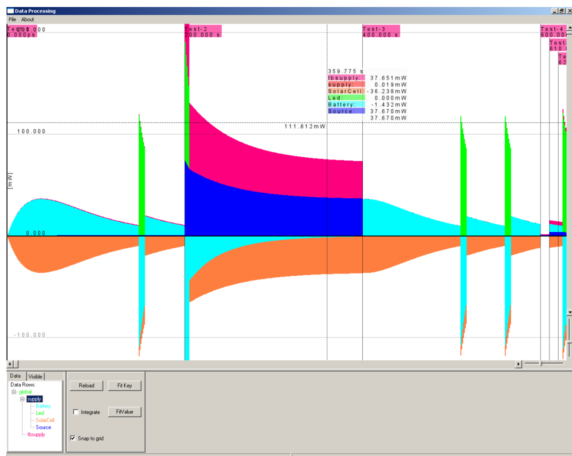

Figure 7: Screenshot of the DataProcessing tool.

6.7 Interrupts

To model interrupts with the PAWiS Framework the interrupt source has to register with `registerIntSource(`*intSource&*`)`. The parameter is a reference to a `TaskControl::IntSource` object. Use its `setName()` method before you register the source to name the interrupt source as later used by the CPU interrupt mapping. Note that this must be called within your `onInit()` method (see Lst. 4).

To issue an interrupt request, the method `TaskControl::intRequest(`*intSource&*`)` is used (see `countTimer()` in Lst. 4). This method is non-blocking, i.e. it immediately returns and the interrupt request is handled by the next coroutine yield point.

To process an interrupt request, an interrupt service routine (ISR) must be registered with `registerIsr(`*isrTask*`,` *intVector*`,` *name*`)` (see `onInit()` in Lst. 3). A separate task context is created for this ISR method (see `App::onTimer()` in Lst. 3) whenever it is executed. The ISR has finished when you return from the method.

It is important to note that the PAWiS Framework separates between interrupt sources and interrupt vectors. Several *interrupt sources* can be mapped to a single vector (e.g. several timer events share one interrupt vector). An interrupt source has a unique name within a node set with the `IntSouce::setName()` method. *Interrupt vectors* are represented by a number (`TaskControl::t_IntVector = int`). For every interrupt vector a single interrupt service routine can be registered.

The mapping from interrupt sources to interrupt vectors is accomplished by the CPU (see Lst. 6). With `setInterruptMapping(`*mapping*`)` a pointer to an `InterruptMapping` object is supplied. This object will be used by the Cpu class to determine the vector number in `registerIntSource()`.

To setup the mapping use `InterruptMapping::setMapping(`*source name*`,` *vector*`)` for every mapping from source to vector (see `onInit()` in Lst. 6).

6.8 Scripting

6.8.1 Usage

The PAWiS framework intrinsically supports scripting with the embedded scripting engine Lua. PAWiS Lua scripts can be used at various occurrences in the execution of a simulation run. A very important aspect is the usage of scripts at network initialization time (i.e., these scripts can be used to setup the network; see Sec. 6.9 for details). Listing 11 shows how a script can be specified in order for being executed at network initialization. The `Config` module can process a parameter named `InitScript` which holds the location of a Lua script file to be executed at simulation startup. This takes place *before* the first module that is derived from `PawisModule` is being initialized.

Listing 11: NED File to reference a Lua init script.

```
// ...

simple Config
  parameters:
    InitScript : string;
endsimple

module MyNetwork
  submodules:
    myConfig : Config
      parameters:
        InitScript = "config.lua";
end
```

The supplied script will be executed immediately. In case a script contains a function named `init` this function will be executed right after the script finishes execution. The script in Lst. 12 shows the usage and consequence of an init-function. When the script is initialized the variable `MyVar` is first created and a the value 17

is assigned. Right after that the variable is increased by one. After the script execution the function `init` is called where `MyVar` is set to 200 via the local variable `MyLocalVar` (note that all variables in Lua are global except if they are defined as local).

Listing 12: Lua init script.

```
function otherFunc()
  -- use local temp variable
  local MyLocalVar = 200;
  return MyLocalVar;
end

function init()
  MyVar = otherFunc();
end

-- MyVar has global scope (within the script)
MyVar = 17;
MyVar = MyVar + 1;
```

A second way to bind Lua scripts to framework objects is that every module derived from `PawisModule` also supports the `InitScript` parameter. Though the behavior differs a little from the Config module's usage. When an init script is assigned to a node module the script is executed for *every* instance of the module type and the context of the script (the Lua stack) exists as long as the instance exists. This is comparable to multiple threads where each thread has its own stack for execution. An important consequence is that calls within such scripts are only related to *one* instance of a module and only modify the local environment (the Lua stack). Scripts that are bound to an instance of a module are said to have *module context*. For such scripts the framework offers additional module related functionality (see the Doxygen documentation of the class LuaFunctionalState).

6.8.2 Functionality

Several PAWiS functions are also available in Lua script code. Scripts provide easy access to functionality for moving, grouping, creating sensor nodes, etc. They can also access and provide PAWiS functional interfaces. The following section introduces some of the most important possibilities of Lua scripts.

Node Grouping and Movement. The script in Lst. 13 gives a short example on how to group nodes and translate the entire group. For this purpose a *node group* needs to be created that allows nodes to be added. Whenever the node group is moved all its members are moved relative to the origin of the node group.

Listing 13: Lua node grouping and movement script.

```
function createNodes()
  -- create a group with origin (0/0)
  nodeGroup = pawis.createNodeGroup(0, 0);
  -- create 10 nodes and add to group
  for i = 1, 10 do
    pawis.addToNodeGroup(
      nodeGroup,
      pawis.createDefaultNode(
        i, "Node"..i, i * 10, 0
      );
    );
  end
end

function moveNodes(xOffset, yOffset)
  pawis.moveGroup(nodeGroup, xOffset, yOffset);
end
```

The script in Lst. 14 assumes to have *module context* and shows how to periodically update its housing node's position. An init script for a module within a node can be set up with the `InitScript` parameter of the `Config` module or in a Lua script itself with the `pawis.bindLuaInitScript` function. The example `init` function schedules a callback to Lua after one second. When the callback is executed the node is moved (with one millimeter per second to the right) and another callback is scheduled again for one second.

Listing 14: Lua position update script.

```
function updatePosition()
  local timeDelta = pawis.SimTime - lastUpdate;
  -- use the module context to get the node module
  pawis.moveNode(module.Node, 1e-3 * timeDelta, 0);
  lastUpdate = pawis.SimTime;
  -- re-schedule a callback in 1 second
  pawis.scheduleLuaCallback(1.0, updatePosition);
end

function init()
  -- schedule a callback in 1 second
  pawis.scheduleLuaCallback(1.0, updatePosition);
  lastUpdate = pawis.SimTime;
end
```

For a detailed description of Lua to PAWiS framework bindings consult the Doxygen documentation of the classes `LuaPawisState` and `LuaFunctionalState`. The first class covers common functions that are always available where the latter covers functions that are only available in scripts with *module context*.

Handling Functional Interfaces. Lua glue code also supports calling functional interfaces from scripts. The PAWiS framework dispatches such calls and invokes either C++ based code

or other Lua script code. However, parameters of functional interfaces need to have strict types and order. As Lua does not have a notion of strict types (it utilizes dynamic typing) functional interface prototypes have to be specified before they can be invoked. In order to introduce the notion of types the PAWiS framework provides various pseudo types for Lua that can be used to declare functional interface prototypes (i.e., `pawis.int`, `pawis.uint`, `pawis.double`, `pawis.float`, `pawis.bool`, `pawis.message`, ...). It is necessary that the types and order of the parameters exactly match the functional interface specification. In addition to calling interfaces from scripts the framework also supports the invocation of functional interfaces from the opposite side, i.e., calls from C++ to Lua script code.

The script in Lst. 15 shows both scenarios: on one hand it calls a functional interface and on the other hand it provides a function that serves as a functional interface. In the function `init` it declares a functional interface prototype *inc* for the module *Adder* by calling `module.declareInterface` with two integer input parameters and one integer output parameter. After this call the script can invoke this functional interface with the specified parameters using `control.invoke` as shown in the function `increment`. With the call to the `module.bindInterface` function the module's (i.e., the module that is bound to the script) functional interface named *increment* is bound to the Lua function `increment` with the same parameters as before. Again, this function gets the specified parameters in the specified order. When this function is invoked it in turn also invokes another functional interface, i.e., the interface *inc* of the module *Adder* (as declared before) and passes the result back to the caller.

Listing 15: Lua script providing and calling a functional interface.

```
function increment(value, increment)
  return control.invoke(
    "Adder", "inc", value, increment
  );
end

function init()
  -- declare a functional interface prototype
  module.declareInterface(
    "Adder",
    "inc",
    {pawis.int, pawis.int},
    {pawis.int}
  );

  -- bind Lua function to functional interface
  module.bindInterface(
    "increment",
    increment,
    {pawis.int, pawis.int},
    {pawis.int}
  );
end
```

6.9 Environment

The *Environment* holds all nodes, obstacles and global properties in a PAWiS simulation. It can be seen as the container (or the current state) for everything that is to be simulated. Most importantly it holds all the sensor nodes on 3D positions and properties for wireless communication among them.

6.9.1 Environment Definition

Node placement. The PAWiS framework offers multiple possibilities to instantiate and configure a simulation network. The OMNeT way to instantiate nodes is to use the NED file (see Sec. 6.1.1). The example NED file in Lst. 16 shows how to instantiate two nodes. First of all the modules (simple and compound) need to be defined. The module that represents a sensor node is a compound module consisting of various submodules that are representing layers or aspects of your system. Then the submodules of the network (these are the actual nodes, i.e., instances of the node module types) are specified. In this example the variables holding the nodes are called node1 and node2 both of the type MyNode. In order for a node module to be PAWiS conform it needs to specify the parameters `NodeId`, `PosX`, `PosY` (`PosZ` is optional and set to zero if omitted).

Listing 16: NED file example for instantiating two nodes.

```
simple Config
  parameters:
    InitScript : string,
    AttenuationExponent : numeric,
    BackgroundNoise : numeric,
    AntennaArea : numeric,
    MinReceivingPower : numeric,
    NodeClass : string;
endsimple

module MyNode
  // ...
endmodule

// ...

module MyNetwork
  submodules:
    myConfig : Config
      parameters:
```

```
        InitScript = "config.lua",
        AttenuationExponent = 2.0,
        BackgroundNoise = 0.0,
        AntennaArea = 0.001,
        MinReceivingPower = 1e-15,
        NodeClass = "MyNode";
    node1 : MyNode
      parameters:
        NodeId = 1,
        PosX = 100.0,
        PosY = 100.0,
        PosZ = 0.0;
    node2 : MyNode
      parameters:
        NodeId = 2,
        PosX = 100.0,
        PosY = 200.0,
        PosZ = 0.0;
  // ...
endmodule

network myNetworkInstance : MyNetwork
endnetwork
```

The NED file also shows the usage of the Config module. This module can be utilized to set up environment properties such as:

AntennaArea: Parameter is used by the Air to calculate the received signal power (see (2)). It is used for every *AirClient* module within the Environment.

AttenuationExponent: Parameter gives the attenuation exponent b in (1) and (2).

BackgroundNoise: Minimal noise that is always effective when sending via the Air.

InitScript: Path to a script file that is executed when the network is initialized.

MinReceivingPower: Threshold for accepting a signal as message (every signal with power below this threshold is treated as noise).

NodeClass: Default class (module type) for instantiating node modules.

PositionFactor: Parameter is used for scaling world positions (in meters) to screen pixels.

An alternative and very flexible way to specify a network is to use a Lua script. In order to utilize this feature a network initialization script hook (as described in Sec. 6.8) needs to be installed. Listing 17 shows an example `init` function of such a script. This script creates 100 nodes and places them on a 10 by 10 meters grid. At the beginning of the function some environment properties necessary for wireless communication are configured (note that these parameters can also be changed during the simulation runtime to simulate changing communication conditions). Scripting supports the same properties as the Config module from the NED file.

Listing 17: Lua script example for setting up a network.

```
function init()
  pawis.AttenuationExponent = 2.0;
  pawis.BackgroundNoise = 0.0;
  pawis.AntennaArea = 0.001;
  pawis.MinReceivingPower = 1e-15;
  pawis.NodeClass = "MyNode";
  local id = 1;
  for y = 1, 10 do
    for x = 1, 10 do
      local aNode =
        pawis.createDefaultNode(
          "Node"..id,
          id,
          x, y
        );
      id = id + 1;
    end
  end
end
```

Using scripts to set up networks allows for complex spatial distributions of nodes (note that topology can not be setup; it is a mere result of signal power between senders and receivers).

Attenuation. The attenuation between the sensor nodes is mainly calculated by their distance. Currently no obstacles are considered (see Sec. 5.3.8). In order to adjust the attenuation between two nodes the PAWiS framework offers the possibility to manually specify a multiplicative factor for attenuation between pairs of nodes. Whenever such an individual attenuation is set up the attenuation that results from the distance between two nodes is multiplied with this factor. The Lua script in Lst. 18 creates 10 adjacent nodes in a function called `setupNodes`. Right after creating the nodes the attenuation between nodes 1 and 2 and the attenuation between nodes 2 and 3 is adjusted with the call to `pawis.setAttenuation`. The function takes a pair of node IDs and an attenuation factor that is a multiplicative factor (i.e., values above 1 add attenuation while values below 1 reduce attenuation) for power.

Listing 18: Lua script example for specifying custom attenuation between nodes.

```
function setupNodes()
  local id = 1;
  for x = 1, 10 do
    local aNode =
      pawis.createDefaultNode(
        "Node"..id,
```

```
      id, x, 0
    );
  id = id + 1;
end
pawis.setAttenuation(
  {
    {1, 2, 2.0},
    {2, 3, 2.0}
  }
);
end
```

6.10 Air

The module of your node which connects to the Air must be derived from `AirClientModule`.[3] The transmitter uses the method `sendToAir(`*message*`,` *bit count*`,` *transmit power*`,` *transmission ID*`)` to transfer the data to the Air. The parameter *message* is the message as an object which is derived from `BaseMessage`. The number of bits in the packet is specified by *bit count*, which will be used for calculating the duration. The *transmit power* is given in Watts. The meaning of *transmission ID* is described below. When the packet transmission is complete, the callback method `void onAirDataTransmitted()` is executed.

On the receiving side at the start of a transmission the callback method `bool acceptAirDataStart(`*signal power*`, &`*preview bits*`,` *transmission ID*`)` is called. This should return `true` if the packet will be accepted (i.e. the receiver is listening). During the reception of the packet no new packets can be received, therefore `false` must be returned. Obviously when the radio is not in listen state `false` should be returned too. The *signal power* at the receiver is given in Watts.

When the packet reception is finished, the callback method `onAirDataArrived(`*message*`,` *bit count*`,` *bit errors*`)` is executed. It should implement the actual data packet handling. *message* and bit count are the message object and the value supplied to `sendToAir()`. The number of bit errors which were introduced to the packet during transmission are given in *bit errors*. Note that *message* is unaltered (i.e. without errors). Only the number of bit errors is given and you don't know *which* bits of the message were disturbed.

Since the method `onAirDataArrived()` is only executed after the full packet was received, no notification is provided while the transmission is in progress. For some addressing schemes or special receivers this might be necessary, therefore the (output) parameter `&`*preview bits* of `acceptAirDataStart()` should be set to the number of bits, after which you want to be notified. The virtual callback method `onPreviewPacket()` will be executed. Set it to `0` if you don't want an additional notification.

If during the transmission another node starts to send, its signal is uncorrelated to the first sender. This can be modeled as noise and therefore decrease in SNR (from the receivers point of view). Such events can happen several times during the transmission of a data packet and the receiver has to deal with changing SNR throughout the packet. The final count of bit errors thus results from this train of different SNR values and is assembled of the portions of constant SNR.

The number of bit errors within a period of constant SNR is calculated by the callback method `unsigned int calcBitErrors(`*SNR*`,` *bit count*`)`. It should use the given SNR value and calculate the BER and further the number of bit errors.

The simulation of a shared channel via multiple access schemes like TDMA, FDMA and CDMA is enabled in two different ways. The case of TDMA is trivial, because we already have a time domain simulation. FDMA and CDMA, on the other hand, occupy the medium concurrently and are only separated by a frequency channel or by coding groups. To implement this behavior the *transmission ID* parameter of the methods `acceptAirDataStart()` and `sendToAir()` is used. This is an `unsigned int` with no internal meaning. You define which value represents which channel (or whatever else you like), hence it is just an ID.

When sending a packet with `sendToAir()` simply specify the appropriate ID (e.g. to state the frequency channel or the coding group). All other nodes get notified by the method `acceptAirDataStart()` which also supplies the ID. In this function you define how different IDs are handled. In Lst. 19 we only accept the packet if it is sent with the same ID as our own (`OwnTID`).

If a node doesn't accept a packet the signal is again regarded as noise. Usually an input filter of the receiver attenuates signals at neighbor channels. This is modeled by the method `calcInterferingNoise()`. This is the place where you specify the input filter characteristics (side channel suppression, ... for FDMA) or the coding gain (for CDMA). The return value is a factor which is multiplied by the received power, hence its return value should be $\leq$ 1.0. If you don't care about the channel ID, simply return 1.0 and don't handle the *transmission ID* in `acceptAirDataStart()`.

Listing 19 gives an example of a physical layer implementation. The function `calcBitErrors()` calculates the absolute amount of bit errors for a

[3] This is derived from `PawisModule` so your module is all like a normal module.

given SNR pa_fSnr and bit count pa_nBitCount. The bit error ratio (BER) is derived from the SNR by the function calc_BER() which depends on the modulation format. It is not shown here. The number of bit errors is binomially distributed and calculated with the OMNeT++ function binomial(n,p).

The function onAirDataStart() demonstrates how the return value depends on the internal state of the receiver. If the signal power is below a certain sensitivity level the receiver cannot synchronize to the RF signal and thus doesn't receive the signal. If the signal is sent on the wrong channel the packet is also not accepted.

onAirDataArrived() is executed when the packet reception is done. This will update the internal transceiver state and forward the message to the upper layers.

The upper layers have to set the physical layer into listen mode, which is done with the Phy::listen() functional interface. It first activates the receiver (by setting its state to rsListen. Then it waits until a packet is received or a timeout is reached. A status and the received packet are then returned to the caller (i.e. the MAC layer).

To send a packet the functional interface Phy::send() is invoked from an upper layer. This sets the transceiver state and uses sendToAir() to start the actual transmission. As soon as it has finished the callback onAirDataTransmitted() sets the transceiver state back to idle and informs the upper layers of the completed transmission. Note that this is stateless and the integrity is not ensured. This has to be accomplished by upper layers.

Listing 19: Implementation of the physical layer “Phy” interfacing to the Air from phy.cpp.

```
/* ... */

/**** Interface to the Air ****/
unsigned int Phy::calcBitErrors(
  double SNR,
  unsigned int BitCount)
{
  // ...
  BER    = calcBER(SNR);
  Errors = binomial(BitCount, BER);
  return BitErrorCount;
}

double Phy::calcInterferingNoise(
  unsigned int TransmissionID)
{
  return calcNeighChan(OwnTID, TransmissionID);
}

bool Phy::acceptAirDataStart(
  double SignalPower,
  int &PreviewBits,
  unsigned int TransmissionID)
{
  PreviewBit = 0;
  switch (RadioState) {
    case rsListen:
      if ((SignalPower > MinPower) &&
          (TransmissionId == OwnTID)) {
        RadioState = rsReceive;
        return true;
      } else
        return false;
    case rsReceive:
    default:
      return false;
  }
}

void Phy::onAirDataArrived(
  BaseMessage *Data,
  unsigned int BitCount,
  unsigned int BitErrors)
{
  // handle radio state
  RadioState = rsFSON;
  // inform MAC layer
  // ...
}

void Phy::onAirDataTransmitted(void) {
  // handle radio state
  RadioState = rsFSON;
  // inform MAC layer
  // ...
}

/**** Functional Interfaces ****/
void Phy::listen(TaskControl &pa_oControl) {
  // ...
  RadioState = rsListen;
  // ...
}

void Phy::send(TaskControl &pa_oControl) {
  // ...
  RadioState = rsSend;
  sendToAir(txData.dup(), txData.getLength()*8,
    m_dOutputPower, OwnTID);
  // ...
}

/* ... */
```

6.11 Summary

6.11.1 Tasks

Overview of tasks

- Task
  - FI: single invoke
  - FI: multi invoke

- Call Back
- Predicate functions

6.11.2 Casts

The following casts are available

CAST_TASK() is used for `registerFunctionalInterface()` and `startTask()` to cast a task method.

CAST_CALLBACK() is used for `scheduleCallback()`.

CAST_PREDICATE() is used with `requireCpuUntil()`, `requireCpuOrUntil()` and `waitUntil()`.

7 Important Notes

7.1 OMNeT++ Messages

- When directly using OMNeT++ messages the *owner* is responsible to `delete` the message. This is not always the creator!
- Never send OMNeT++ messages directly, only use them as data type.
- When a message is supplied to a task (e.g., with `invoke()`) the called task is the new owner of the message. After return from this task the input parameters are invalid. To return values use the output parameters. The input parameters are automatically cleared at the return of `invoke()`.
- When a message is supplied to a method of the framework, the framework is the new owner of this message and thus also responsible to `delete` it.

7.2 Software and hardware tasks

- Software tasks must not `invoke()` hardware tasks and vice versa. This is checked by the framework and leads to a runtime error. Nonetheless, software tasks can start a hardware task with `startTask()`

8 Patterns

8.1 Non-blocking functional interfaces

To implement a functional interface which returns immediately but starts a parallel action (e.g. starting an ADC conversion), use the following approach. Write the code which runs in parallel in a separate task method. In your functional interface code use `startTask()` to start that separate task. When this has done its job simply quit the method (e.g. at its end or with `return`.

8.2 More to come

References

[GSS+04] Y. Gsottberger, X. Shi, G. Stromberg, W. Weber, T.F. Sturm, H. Linde, E. Naroska, and P.; Schramm. Sindrion: a prototype system for low-power wireless control networks. In *IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, pages 513 – 515, 25-27 October 2004.

[H+03] Robert F. Heile et al. 802.15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs). Technical report, IEEE Computer Society, 1. October 2003.

[IEE03] *IEEE Std. 802.15.4-2003, IEEE Standard for Information Technology - Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks (WPANs)*, 2003.

[MB04] Stefan Mahlknecht and Michael Böck. CSMA-MPS: A Minimum Preamble Sampling MAC Protocol for Low Power Wireless Sensor Networks. *WFCS*, 2004.

[Roe04] Matthias Roetzer. Routing in energieautarken Funksensornetzwerken. Master's thesis, Vienna University of Technology, 2004.