A Comprehensive Approach for Power Management in Embedded Systems

Journal:	*Transactions on Embedded Computing Systems*
Manuscript ID:	TECS-2009-0181
Manuscript Type:	Paper
Date Submitted by the Author:	10-Nov-2009
Complete List of Authors:	Fröhlich, Antônio Augusto; UFSC, LISHA
Computing Classification Systems:	\category{D.4.7}{Organization and Design}{Real-time systems and embedded systems}, \terms{Design, Management, Experimentation} , \keywords{Embedded systems, power management, energy-aware scheduling}

A Comprehensive Approach for Power Management in Embedded Systems

ANTÔNIO AUGUSTO FRÖHLICH
Federal University of Santa Catarina

In this article, power management is addressed in the context of embedded systems from energy-aware design to energy-efficient implementation. A set of mechanisms specifically conceived for this scenario is proposed, including: a power management API defined at the level of user-visible system components, the infrastructure necessary to implement the API (namely, battery monitoring, accounting, auto-suspend, and auto-resume), an energy-event propagation mechanism based on *Petry Nets* and implemented with *Aspect-Oriented Programming* techniques, and an autonomous power manager based on the proposed API and infrastructure. The proposed mechanisms are illustrated and evaluated by realistic embedded systems that enable the proposed mechanisms to be compared with other proposals at each of the considered levels. As a result, this article has its main contribution on the introduction of a comprehensive and systematic way to deal with power management issues in embedded systems.

Categories and Subject Descriptors: D.4.7 [**Organization and Design**]: Real-time systems and embedded systems

General Terms: Design, Management, Experimentation

Additional Key Words and Phrases: Embedded systems, power management, energy-aware scheduling

1. INTRODUCTION

Power management is a subject of great relevance for two large groups of embedded systems: those that operate disconnected from the power grid, taking their power supply from batteries, photovoltaic cells, or from a combination of technologies that yet impute limitations on energy consumption; and those that face heat dissipation limitations, either for relying on high-performance computations or for being embedded in restrictive environments. Both classes of embedded systems can benefit from power management techniques at different levels, from energy-efficient peripherals (e.g. sensors and actuators), to adaptive digital systems, to power-aware software algorithms.

Historically, power management techniques rely on the ability of certain components to be turned on and off dynamically, thus enabling the system as a whole to save energy when those components are not being used [Monteiro et al. 1996]. Only more recently, techniques have been introduced to enable some components to

Author's address: Antônio Augusto Fröhlich, Laboratory for Software/Hardware Integration, Federal University of Santa Catarina, 88040900 Florianópolis, SC, Brazil, guto@lisha.ufsc.br.

operate at different energy levels along the time [Benini et al. 1998]. Multiple operational modes and *Dynamic Voltage Scaling* (DVS) are examples of such techniques that are becoming commonplace for microprocessors. Unfortunately, microprocessors are seldom the main energy drain in embedded systems—peripherals are—, so traditional on/off mechanisms are sill of great interest.

Even concerning microprocessors, which are the cores of the digital systems behind any embedded system, current power management standards, such as APM and ACPI, only define a software/hardware interface for power management, mostly disregarding management strategies and fully ignoring the designer knowledge about how energy is to be used—and therefore how it can be saved—in the system. Moreover, these standards evolved in the context of portable personal computers and usually do not fit in the limited-resource scenario typical of embedded systems. Other initiatives in the scope of embedded operating systems, some of which will be discussed later in this article, introduce power management mechanisms at the level of hardware abstraction (viz. HAL), what forces programmers to go down to that level in order to manage energy, thus compromising several aspects of software quality, in particular, portability. Yet, others assume that the operating system is capable to doing power management by itself.

We believe that power management in embedded systems can be made far more effective if adequate means are provided so that designers can express their knowledge about the power characteristics of the system directly to the power manager. In contrast to generic-purpose systems, embedded systems are designed for specific purposes, most often entangled in a hardware/software project driven by the requirements of a single application. Assuming that the traditional autonomous power management mechanisms found in portable computers will ever be able to match the designers expertise about such tailor-made systems is unrealistic.

In this article, we introduce a set of mechanisms that enable designers to directly influence, and even control, the power management strategy for the system. These mechanisms have been modeled around typical embedded system requirements, including small foot-print, little overhead and non-interference in real-time constraints. They are:

—A power management API defined at the level of user-visible system components (e.g. files, sockets and processes) that supports semantic energy modes (i.e. *off*, *stand-by*, *light* and *full*), arbitrary energy modes (i.e. device-specific), and dynamic voltage scaling.

—A power management infrastructure for system components, with accounting, auto-suspend and auto-resume mechanisms, implemented around *Aspect-Oriented Programming* (AOP) concepts and formalized through *Petry Nets*.

—An autonomous power manager, whose policies can be configured, statically or dynamically, and whose decisions take in consideration the interactions between applications and system done through the management API, thus enabling applications to override specific policies.

The reminder of this text discusses the design of these three mechanisms, their implementation for the EPOS Project, experiments carried out to corroborate the proposed mechanisms, a discussion about related work and is closed with a reasoning about the proposed mechanisms.

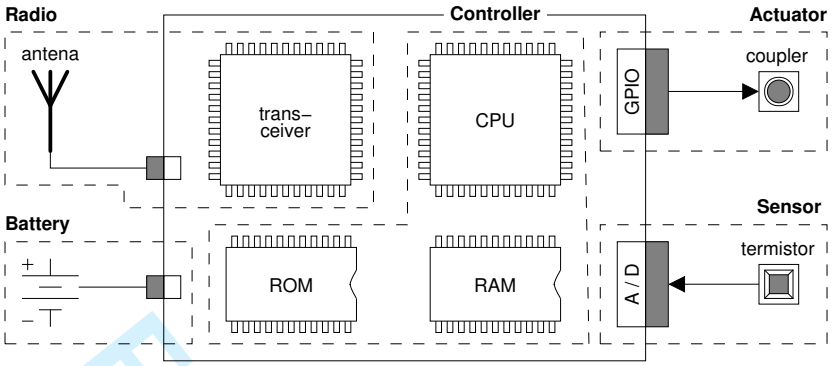

Fig. 1. Example monitoring system block diagram.

2. POWER MANAGEMENT API

In order to introduce a discussion about power management *Application Program Interfaces* (API), let us first recall how energy consumption requirements arise during the design of an energy-aware embedded system and how they are usually captured. In such systems, designers look for available energy-efficient components and, eventually, specify new components to be implemented. During this process, they inherently acquire knowledge about the most adequate operating strategy for each component and for the system as a whole. Whenever the identified strategies are associated to modifications in the energy level of a given component, this can be captured in traditional design diagrams, such as sequence, activity and timing, or by specific tools [Chou et al. 2002].

Now let us consider the design of a simple application, conceived specifically to illustrate the translation of energy constraints from design to implementation. This application realizes a kind of remote monitoring system, capable of sensing a given property (e.g. temperature), reporting it to a control center, and reacting, by activating an actuator (e.g. external cooler), whenever it exceeds a certain limit. Interaction with the control center is done via a communicator (e.g. radio). The system operates on batteries and must run uninterruptedly for one year. A block diagram of the system is shown in figure 1.

The application is modeled around four threads whose behavior is depicted in the sequence diagrams of figures 2 through 5: `Main` allocates common resources and creates the working threads; `Monitor` is responsible for periodic temperature monitoring (every second), for reporting the current temperature to the control center (every 10 seconds), and, in case the temperature threshold is exceeded, for triggering the emergency handling thread; `Trigger` is responsible for triggering the emergency handling thread on command of the control center; and `Recovery`, the emergency handling thread, is in duty of controlling an actuator intended at restoring the temperature to its normal level. Coordination is ensured by properly assigning priorities to threads and by the `emergency` semaphore.

In the sequence diagrams of figures 2 through 5, energy-related actions captured during design are expressed by messages and remarks. For instance, the knowledge that the `Thermometer` component uses a low-cost thermistor and therefore must

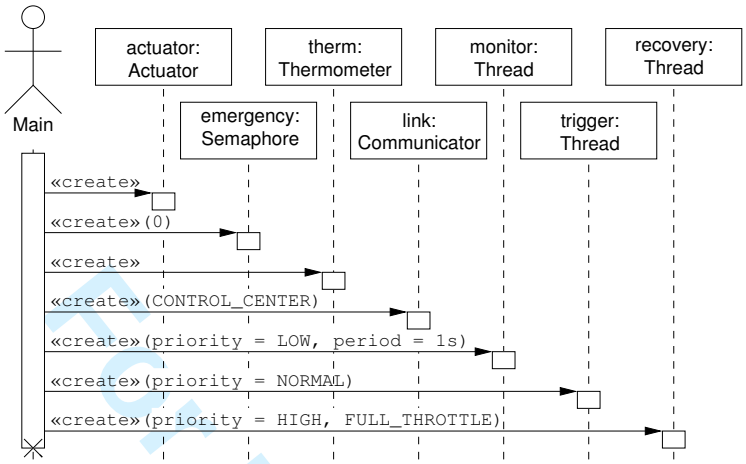

Fig. 2. Main thread sequence diagram with power management actions.

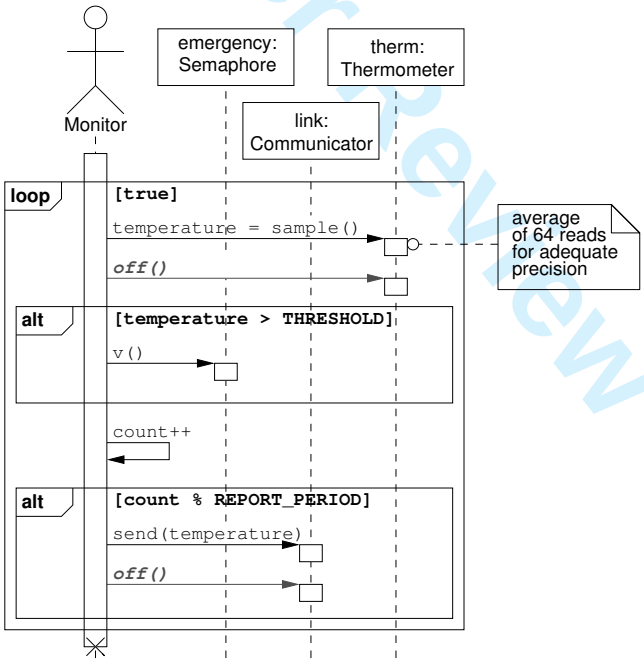

Fig. 3. Monitor thread sequence diagram with power management actions.

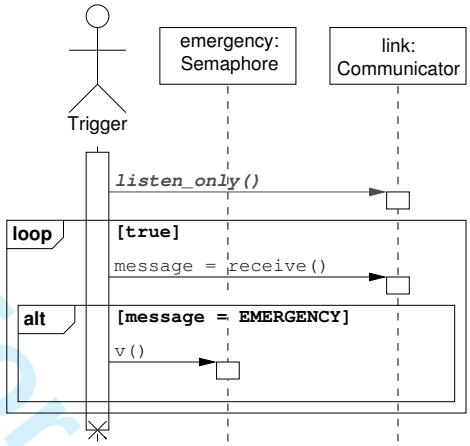

Fig. 4. Trigger thread sequence diagram with power management actions.

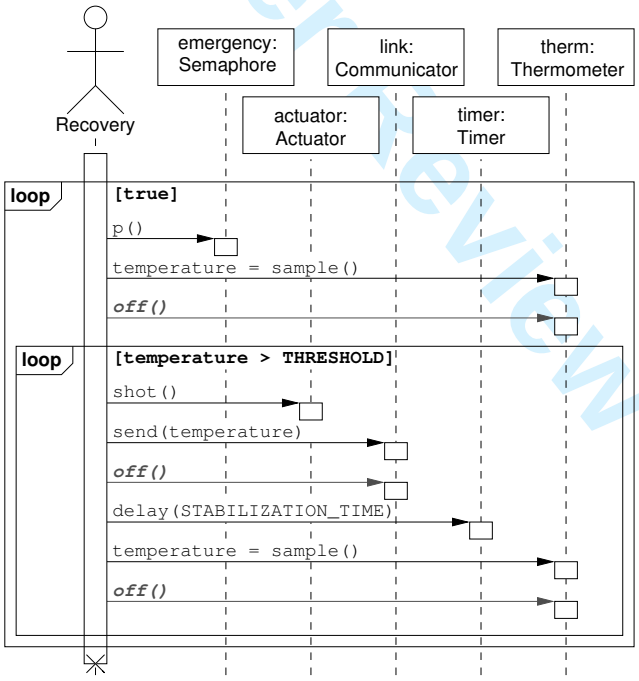

Fig. 5. Recovery thread sequence diagram with power management actions.

Table I. Example monitoring system energy consumption estimates.

Block	State	Cur. (μA)	Duty Cycle	Cons. (μAh)	Cons. (mAh)	Energy (J)
Controller	stand-by	8	98.69%	7.90	0.0344	0.3720
	idle	470	0%	0.00		
	min. freq.	1,950	1.30%	25.35		
	max. freq.	12,000	0.01%	1.20		
Radio	off	0.2	0.20%	0.00	0.3722	4.0199
	listen	96	97.78%	93.87		
	receive	9,600	$<$ 0.01%	0.96		
	transmit	13,800	2.01%	277.38		
Sensor	off	1	99.97%	1.00	0.0014	0.0147
	at 25 C	900	0.02%	0.18		
	at 40 C	1,775	$<$ 0.01%	0.18		
Actuator	on	2,200	$<$ 0.01%	0.22	0.0002	0.0024
	off	0	99.99%	0.00		
Total consumption per hour					**0.4082**	**4.4089**

perform 64 measurements before being able to return a temperature value within the desired precision is expressed as a note associated with the method (figure 3). In order to be energetically efficient, the circuitry behind `Thermometer` (i.e. ADC and thermistor) should be kept active during all the measurement cycle, thus avoiding repetitions of electrical stabilization phase[1]. It can be turned off short after.

The diagrams also show power management hints for the `Communicator` component (figures 3, 4, and 5), which is mostly used to listen for a message from the control center and thus can be configured on a listen-only state for most of the time. And, indirectly, also for the `CPU` component, which must operate in the maximum frequency while running the `Recovery` thread, but can operate in lower frequencies for the other threads (figure 2). With this information in hand, the system can be implemented to be far more efficient in terms of energy, either by the programmer himself or by means of a automatic power manager.

Considering the functional properties described so far and the threads execution frequencies, it is possible to estimate duty cycles for each of the major components in our example. This information can then be combined with energy consumption estimates of individual components to calculate the power supply required by the system. This procedure is summarized in table I, which shows hypothetical energy consumption estimates, duty cycles and energy consumption for the four major components in the system[2].

In order to match the requirement of operating uninterruptedly for one year, the system would demand a battery capable of delivering approximately 3576 mAh (at 3 V). For comparison, the same system operating with all components constantly active, that is, without any power saving strategy, would require about 142 Ah, almost 40 times more than in our example.

[1] In a typical ADC/thermistor configuration, the electrical stabilization phase accounts up to 98% of the measurement cycle, both in terms of time and in terms of energy [Panasonic 2004].
[2] Estimates were based on the Mica2 sensor node [Hill et al. 2004]

```
int thermometer_sample() {
    // Open ADC power attribute on sysfs
  struct sysfs_attribute * adc_power
    = sysfs_open_attribute(
        "/sys/devices/i2c/AD8493/power/state"
      );

  int accumulator = 0;
  int value;

  // Switching to power state 0 (ON)
  sysfs_write_attribute(adc_power, "0", 1);
  for(int i = 0; i < 64; i++) {
      // Read ADC result
    ad8493_get(adc_device, &value);
    accumulator += value;
  }
  // Switching to power state 3 (OFF)
  sysfs_write_attribute(adc_power, "3", 2);

  // Convert reading into celcius degrees
  return raw_to_celcius(accumulator / 64);
}
```

Fig. 6. `Thermometer::sample()` method implementation for μCLINUX.

2.1 Current APIs

Few systems targeting embedded computing can claim to deliver a real Power Management API. Nevertheless, most systems do deliver mechanisms that enable programmers to directly access the interface of some hardware components. These mechanism, though not specifically designed for power management, can be used for that purpose at the price of binding the application to hardware details.

μCLINUX, like many other UNIX-like systems, does not feature a real power management API. Some device drivers provide power management functions inspired on ACPI. Usually these mechanisms are intended to be used by the kernel itself, though a few device drivers export them via the `/sys` or `/proc` file systems, thus enabling applications to directly control the operating modes of associated devices.

The source code in figure 6 is a user-level implementation of the `Thermometer::sample()` method of our example monitoring system. In this implementation, programmers must explicitly identify the driver responsible for the ADC to which the thermistor is connected. Besides the overhead of interacting with the device driver through the `/sys` file system, μCLINUX PM API creates undesirable dependencies and would fail to preserve the application in case the thermistor gets connected to another ADC channel or in case the ADC in the system gets replaced by another model.

TINYOS, a popular operating system in the wireless sensor network scene, allows programmers to control the operation of hardware components through a low-level, architecture-dependent API. Though not specifically designed for power manage-

```
// ...
  // When initializing system
  event void Boot.booted() {
    // ...
    // Put radio in listening mode
    call RadioControl.start();
    // ...
  }

  // When data is received
  event message_t* Receive.receive(
                      message_t* bufPtr,
                      void* payload, uint8_t len
                    ) {
    if (len != sizeof(radio_sense_msg_t))
      return bufPtr;
    else {
      radio_sense_msg_t* rsm =
            (radio_sense_msg_t*) payload;
      if (rsm->data == EMERGENCY) {
        // Turn radio off
        // Someone has to turn it on again later
        RadioControl.stop();
        emergency_semaphore++;
      }

      return bufPtr;
    }
  }
//...
```

Fig. 7. `Trigger` thread implementation for TinyOS.

ment purposes, this API ensures direct access to the hardware and thus can be used in this sense. When compared to μCLinux, TinyOS delivers a lighter mechanism, more adequate for most embedded system, yet suffers from the same limitations as regards usability and portability. The use of TinyOS hardware control API for power management is illustrated in figure 7, which depicts the implementation of the `Trigger` thread of our example.

Mantis, features a Posix-inspired API that abstracts hardware devices as Unix special files. Differently of μCLinux and TinyOS, however, Mantis does not propose that API to be used for power management purposes: internal mechanisms automatically deactivate components that have not been used for a given time, or perform an "on-act-off" scheme, thus implementing a sort of OS-driven power manager. This strategy can be very efficient, but makes it difficult for programmers to express the knowledge about energy consumption acquired during the design phase. This is made evident in the implementation of the `Thermometer::sample()` depicted in figure 8. Unaware of the precision required for the temperature vari-

```
for (int i = 0; i < 64; i++) {
  // Read from device
  dev_read (DEV_MICA2_TEMP, &data, 1);
  accumulator += data;
}
// MantisOS device driver turns sensor
// ON and OFF for every reading
```

Fig. 8. Thermometer::sample() method implementation for MANTIS.

able, MANTIS cannot predict that the ADC is being used in a loop and misses the opportunity to avoid the repetition of the expensive electrical stabilization phase of the thermometer operation.

Some systems assume that architectural dependencies are intrinsic to the limitations of typical embedded systems, however, this is exactly the share of the computing systems market that could benefit from a large diversity of suppliers [Tennenhouse 2000] and therefore would profit from quick changes from one architecture to another. This, in addition to the fact that current API does not efficiently support the expression of design knowledge during system implementation, led us to propose a new PM API.

2.2 Proposed API

The Power Management API proposed here arose from the observation that currently available APIs does not fit in the models of embedded systems produced during design and also induce undesirable architectural dependencies between the embedded application and the initial hardware platform. In order to overcome these limitations, we believe a PM API for embedded systems should present the following characteristics:

—Support direct application control of energy-related issues, yet without excluding cooperation with an automatic power manager;

—Act also at the level of user visible components, instead of being restricted to the interface of hardware components, thus promoting portability and usability;

—Be suitable for both application and system programming, thus unifying power management mechanisms and promoting reuse;

—Include, but not be restricted to, semantic modes, thus enabling programmers to easily express power management operations, but avoiding the limitations of a small, fixed number of operating modes (as is the case of ACPI).

With these guidelines in mind, we developed a very simple API, which comprises only two methods and an extension to the methods responsible for process creation. They are:

```
Power_Mode power()
power(Power_Mode)
```

The first method returns the current power mode of the associated object (i.e., component), while the second allows for mode changes. Aiming at enhancing us-

Table II. Semantic power modes of the proposed PM API.

Mode	FULL	LIGHT	STANDBY	OFF
Energy	high	low	very low	none
Functionality	all	all	none	none
Performance	high	low	NA	NA
Wake up Delay	NA	very low	high	very high

ability, four power modes have been defined with semantics that must be respected for all components in the system: *off*, *stand-by*, *light* and *full*. Each component is still free to define additional power modes with other semantics, as long as the four basic modes are preserved. Enforcing universal semantics for these power modes enables application programmers to control energy consumption without having to understand the implementation details of underlying components (e.g., hardware devices). Allowing for additional modes, on the other hand, enables programmers to precisely control the operation of special component, whose operation transcend the predefined modes.

The introduction of these methods in user-visible components certainly requires some sort of propagation mechanism and could itself introduce undesirable dependencies. We will describe an strategy to implement them using a combination of *Aspect-Oriented Programming* techniques and *Hierarchical Petry Nets* in section 3. For now, lets concentrate on the characterization of the API, not the mechanisms behind it.

Table II summarizes the semantics defined for the four universal operating modes. A component operating in mode *full* provides all its services with maximum performance, possibly consuming more energy than in any other mode. Contrarily, a component in mode *off* does not provide any service and also does not consume any energy. Switching a component from *off* to any other power mode is usually an expensive operation, specially for components with high initialization and/or stabilization times. The mode *stand-by* is an alternative to *off*: a component in *stand-by* is also not able to perform any task, yet, bringing it back to *full* or *light* is expected to be quicker than from mode *off*. This is usually accomplished by maintaining the state of the component "alive" and thus implies in some energy consumption. A component that does not support this mode natively must opt between remaining active or saving its state, perhaps with aid from the operating system, and going off.

Defining the semantics for mode *light* is not so straightforward. A component in this mode must deliver all its services, but consuming the minimum amount of energy. This definition brings about two important considerations. First, if there is a power mode in which the component is able to deliver all its services with the same quality as if it was in mode *full*, then this should be mode *full* instead of *light*, since it would make no sense to operate in a higher consumption mode without arguable benefits. Hence, mode *light* is often attained at the cost of performance (e.g., through DVS). This, in turn, brings about a second consideration: for a real-time embedded system, it would be wrong to state that a component is able to deliver "all its services" if the added latency is let to interfere with the time requirements of applications. Therefore, mode *light* shall not be implicitly propagated to the CPU component. Programmers must explicitly state that they agree to slow

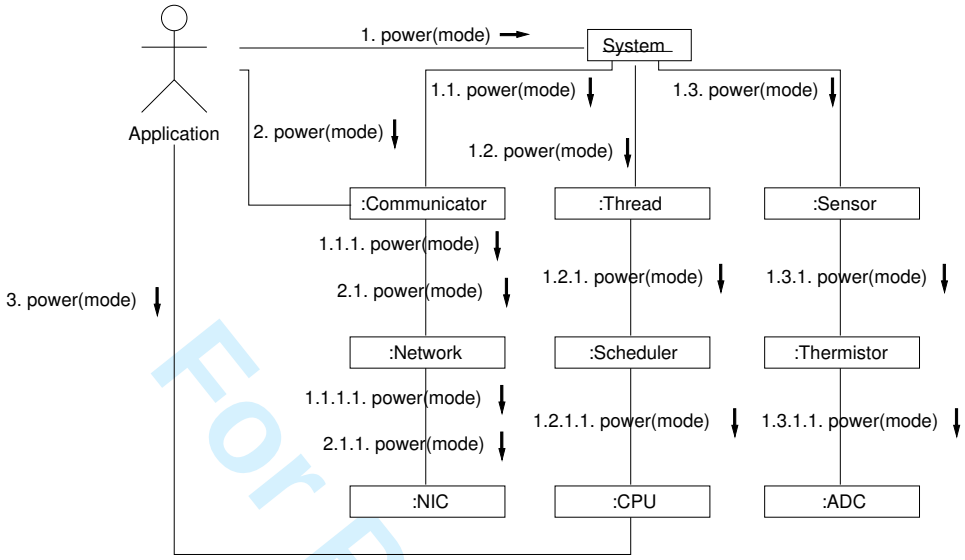

Fig. 9. Power Management API utilization example.

down the processor to save energy, or a energy-aware, real-time scheduler must be deployed [Wiedenhoft et al. 2007].

Besides the four operating modes with predefined, global semantics, a component can export additional modes through the API. These modes are privately defined by the component based on its own peculiarities, thus requiring the client components to be aware of their semantics in order to be deployed. The room for extensions is fundamental for hardware components with many operating modes, allowing for more refined energy management policies.

The proposed API also features the concept of a `System` pseudo-component, which can be seen as a kind of aggregator for the actual components selected for a given system instance. The goal of the `System` component is to aid programmers to express global power management actions, such as putting the whole system in a given operating mode, perhaps after having defined specific modes for particular components.

Figure 9 presents all these interaction modes in a UML communication diagram of a hypothetical system instance. The application may access a global component (`System`) that has knowledge of every other component in the system, triggering a system-wide power mode change (execution flow 1). The API can also be accessed to change the operating mode of a group of components responsible for the implementation of a specific system functionality (in this example, communication functionality through execution flow 2). The application may also access the hardware directly, using the API available in the device drivers, such as *Network Interface Card* (`NIC`), `CPU`, `ADC` (in the figure, application is accessing the CPU through the execution flow 3). The API is also used between the system's components, as can be seen in the figure.

In a system that realizes the proposed API, the monitoring system introduced earlier could be implemented as show in figure 10, a rather direct transcript of the sequence diagrams of figures 2 through 5.

3. POWER MANAGEMENT INFRASTRUCTURE

From the discussion about traditional power management APIs for embedded systems in the previous section, we can infer that the infrastructure behind those APIs are mostly based on features directly exported by hardware components and do not escape the software/hardware interface. As a matter of fact, the power management infrastructure available in modern hardware components is far more evolved than the software counterpart, which not rarely is restricted to mimic the underlying hardware. For example, XSCALE microprocessors support a wide range of operating frequencies that allow for fine grain DVS. They also feature a Power Management Unit, which manages "idle", "sleep" and a set of "quick wake-up" modes, and a Performance Monitoring Unit, which furnishes a set of event counters that may be configured to monitor occurrence and duration of events.

Power management mechanisms may benefit from these features to deploy a context-aware power management policy. Device drivers in the operating systems discussed in the previous section, however, do not make use of most of these features, thus forcing application programmers to develop non-portable, complex, user-level software to develop specific tasks. The same can be observed with peripherals such as wireless network cards, which often provide a large set of configurable characteristics that are not well explored.

Furthermore, even with a well-established standard target exactly at the software/hardware interface level (i.e., ACPI), most embedded operating systems do not implement uniform power management infrastructures, letting the decision about which parts of the standard must be implemented to device driver developers.

In order to allow application-directed or autonomous power management, such an infrastructure would have to feature battery monitoring, accounting, auto-resume, and auto-suspend mechanisms on a component basis.

Battery monitoring:. monitoring battery charge at run-time is important to support power management decisions, including generalized operating mode transitions when certain thresholds are reached; some systems are equipped with intelligent batteries that inherently provide this service, others must tackle on techniques such as voltage variation along discharge measured via an ADC to estimate the energy still available to the sustem.

Accounting:. tracking the usage of components is fundamental to any dynamic power management strategy; this can be accomplished by *event counters* implemented either in software or in hardware; some platforms feature event counters that are usually accessible from software, thus allowing for more precise tracking [Bellosa 2000]; in some systems, for which energy consumption measurements have been carried out on a per-component basis, it might even be possible to perform energy accounting based on these counters.

Auto-resume:. a component that has been set to an energy-saving mode must be brought back to activity before it can deliver any service; in order to relieve

```
Thermometer thermometer;
Actuator actuator;
Semaphore emergency(0);
Communicator link(CONTROL_CENTER);

int main() {
  new Thread(&recovery, HIGH_PRIO, NO_DVS);
  new Thread(&trigger, NORMAL_PRIO);
  new Periodic_Thread(&monitor, LOW_PRIO, 1000000);
  return 0;
}
void monitor() {
  int count = 0;
  while(true) {
    int temperature = thermometer.get();
    thermometer.power(OFF);
    if(temperature > THRESHOLD)
      emergency.v();
    if(++count % 10) {
      link.write(temperature);
      link.power(OFF);
    }
    wait_for_next_cycle;
  }
}
void trigger() {
  while(true) {
    link.power(LISTEN_ONLY);
    if(link.read() == EMERGENCY)
      emergency.v();
  }
}
void recovery() {
  while(true) {
    emergency.p();
    int temperature = thermometer.get();
    thermometer.power(OFF);
    while(temperature > THRESHOLD) {
      actuator.on();
      link.write(temperature);
      link.power(OFF);
      delay(STABILIZATION_TIME);
      temperature = thermometer.get();
      thermometer.power(OFF);
    }
    actuator.off();
  }
}
```

Fig. 10. Example monitoring system implementation using the proposed PM API.

programmers from this task, most infrastructures usually implement some sort of "auto-resume" mechanism, either by inserting mode verification primitives in the method invocation mechanism of components or by a trap mechanism that automatically calls for operating system intervention whenever a inactive component is accessed.

Auto-suspend:. with accounting capabilities in hand, a power management infrastructure can deliver "auto-suspend" mechanisms that automatically change the status of components that are not currently being used to energy-saving modes such as *stand-by* or *off*; however, suspending a component to short after resume it will probably spend more energy than letting it to continue in the original mode, therefore, the heuristics used to decide which and when components should be suspended is one of the most important issues in the field and is now subject to intense research [Ren et al. 2005]; auto-suspend can be implicitly triggered by the accounting mechanisms, but often evolve to a dynamic power manager bound to timing and scheduling operating system services.

Our proposed power management API allows interaction between the application and the system, between system components and hardware devices, and directly between application and hardware. Thus, in order to realize this API, each software and hardware component in our system must be modified to support accounting, auto-resume, and auto-suspend. In this sense, the power management API may be seen as a non-functional property of the system.

3.1 Implementation through Aspect Programs

Aspect-Oriented Programming (AOP) [Kiczales et al. 1997] allows non-functional code (e.g. code to support timeliness, sharing, identification dependability) to be kept separate from from software components, thus improving reusability of the latter [Lohmann et al. 2005].

Energy accounting is easily supported through an aspect program in a component-based system, since it primarily involves adding event counters to components, and adapting the component's methods to use this counters. EPOS [Fröhlich 2001], our test system, supports aspects through a technique called *Scenario Adapters*. Figure 11 shows the structure of a scenario adapter in a UML class diagram. The aspect (`Scenario`) implements code that may be inserted before and/or after the targeted operations through the `enter` and `leave` methods. This mechanism also permits the extension of the targeted component's interface and data structures. In order to do that the new interface methods or fields have to be inserted to the scenario's interface, and the targeted component will have its interface extended due to the inheritance between `Scenario` and `Adapter`. Through such a system, event counters could be added to the `Scenario`, and code to use these counters could be added to the component through the `enter` or `leave` methods.

Auto-resuming a component that has been set to an energy-saving mode may be acomplished by testing and conditionally reseting the component's power mode before each method call. This may be easily implemented through an aspect's `enter` method (Figure 12).

Such a method, however, would incur in unnecessary overhead, as the tests occur even when the component is operational. In order to eliminate this test, the

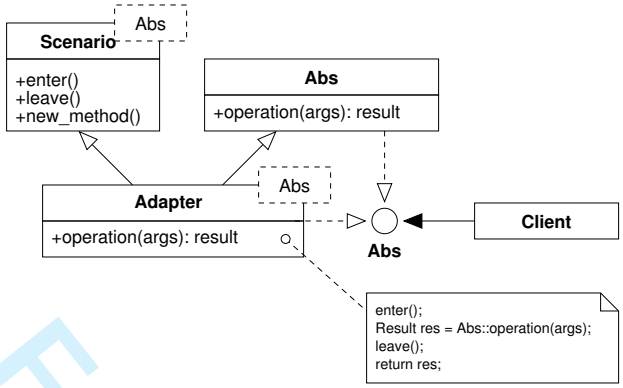

Fig. 11. Epos Scenario Adapter.

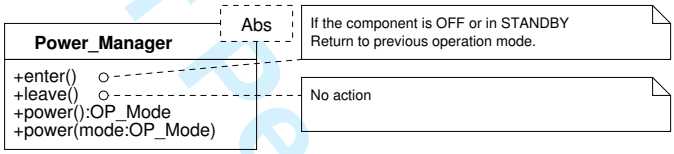

Fig. 12. Auto-resuming aspect

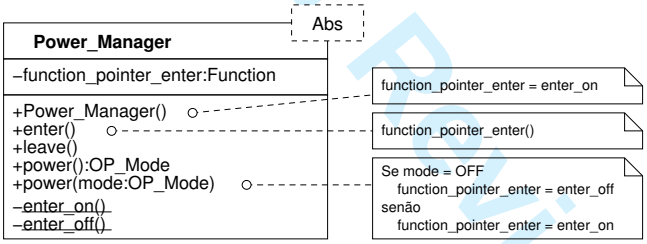

Fig. 13. Auto-resuming aspect using indirect calls

aspect may be used to define and indirect call to each of the component's operation (Figure 13). In this aspect, whenever the power mode is changed, a function pointer to different `enter` methods is updated. When the component is operational (`enter_on`), the aspect takes no action. When the component is stopped, or in standby (`enter_off`), the aspect program ensures that the component is set back to its previous operation mode. This effectively equals the overhead of the aspect to the overhead of the component's `power(Power_Mode)` operation.

Auto-suspend mechanisms also may take advantage of aspect programs, either by using informations collected through the event counters added by aspects, or by implementing an automated suspend policy in the `leave` method.

While it would be possible to define individual aspects to act upon each component in the system, a generalized power management aspect that could be applied to any component in the system would be more interesting. In a power management

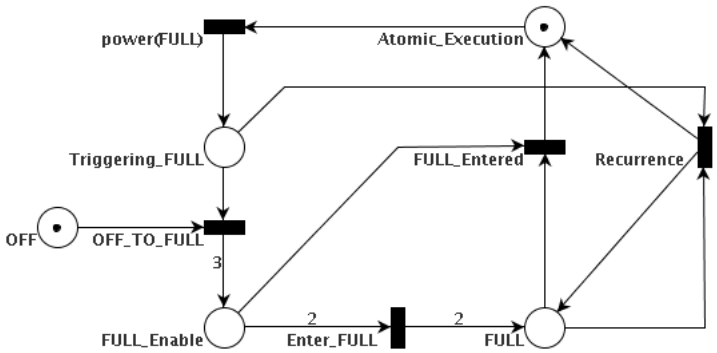

Fig. 14. Generalized Operating Mode Transition Network

aspect, the control of operating mode transitions would have to be generalized in order to be applyable to any component. There is no generic method, however, to implement the transition between different operating modes, as these procedures are dependant to the particular characteristics of different devices, and on the composition of components.

In order to solve this problem, we need to define a mechanism to represent the interactions between components to transition operating modes. One such mechanism, described in the next section, extracts code for implementing operating mode transitions from a set of *Hierarchical Petri Nets* that represent the behavior of components on the operating mode switching process.

3.2 Operation Mode Transition Networks

Petri nets are a convenient tool to model operating mode transitions of components, not only because of its hierarchical representation capability, but also due to the availability of verification tools that can automatically check the whole system for deadlocks and unreachable states [Peterson 1977].

Figure 14 shows a simplified view of the operating mode transition network (only the transition from OFF to FULL is shown). The complete network encompasses all valid transitions in a similar way, with *places* being associated to operating modes (FULL and OFF in the figure), and *resources* designating the component's current operating mode.

The `Atomic_Execution` place is responsible for ensuring that multiple mode change operations do not execute simultaneously. For that, this place is always initialized with one resource. When power management API method is invoked, the corresponding transition is triggered (in the figure, `power(FULL)`) and the resource in the `Atomic_Execution` place is consumed. Additionally a new resource inserted into the `Triggering_FULL` place enables the transactions that remove the resources that marks the component's current operating mode (`OFF`). As the component in the example is in the OFF state, only the `OFF_TO_FULL` transition is enabled. When this transition is triggered, the resource that marked the `OFF` place is consumed, and three resources are inserted into the `FULL_Enable` place. This enables the `Enter_FULL` transition, that is responsible for executing the operations that ac-

tually change the component's power mode. After this transition is triggered, two resources are inserted into the `FULL` place, enabling the `FULL_Entered` transition, which finalizes the process, consuming the final resource in the `FULL_Enable` place, and inserting one resource back into the `Atomic_Execution` place. The entire process results in a resource being removed from the `OFF` place and inserted into the `FULL` place.

In order to avoid deadlocks when a component is requested to switch to its current operating mode (i.e., a component in `FULL` mode is requested to go into `FULL` mode), another transition was added to the model: `Recurrence`. This transition returns the resource removed from the `Atomic_Execution` place in case of recurrence.

The generalized network represents the operating mode transitions from a high level perspective, where the particular characteristics involved in the transition of each component are not specified. However, a refinement process is required in order to allow the inference of the transition procedures from this model. This refinement explores the hierarchical characteristic of Petri nets, which allow an entire network to be replaced by a place or transition in order to model a higher level abstraction and, on the other hand, allow places and transitions to be replaced with sub-networks in order to provide a more detailed model.

4. AUTONOMOUS POWER MANAGER

A considerable fraction of the research effort around power management at software-level has been dedicated to design and implement *autonomous power managers* for general-purpose operating systems, such as WINDOWS and UNIX. Today, battery-operated, portable computers, including notebooks, PDAs, and high-end cellphones, can rely on sophisticated management strategies to dynamically control how the available energy budget is spent by distinct application processes. Although not directly applicable to the embedded system realm, those power managers bear concepts that can be promptly reused in this domain.

As a matter of fact, autonomous power managers grab to a periodically activated operating system component (e.g. timer, scheduler, or an specific thread) in order to trigger operation mode changes across components and thus save energy. For instance, a primitive power manager could be implemented by simply modifying the operating system scheduler to put the CPU in standby whenever there are no more tasks to be executed. DVS capabilities of underlying hardware can also be easily exploited by the operating system in order to extend the battery lifetime at the expense of performance, while battery discharge alarms can trigger mode changes for peripheral devices [Devadas and Aydin 2008]. Nevertheless, these basic guidelines of power management for personal computers must be brought to context before they can be deployed in embedded systems:

—Embedded systems are often engineered around hardware platforms with very limited resources, so the power manager must be designed to be as slim as possible, sometimes taking software engineering to its limits.

—Many embedded systems run real-time tasks, therefore a power manager for this scenario must be designed in such a way that its own execution does not compromise the deadlines of these tasks. Furthermore, the decisions taken by an autonomous power manager must be in accordance with the requirements

of such tasks, since the latency of operating mode changes (e.g. waking up a component) may impact their deadlines. For a real-time embedded system, having a power manager that runs unpredictably might be of consequences similar to the infamous garbage collection issues in JAVA systems [Bacon et al. 2003].

—Embedded systems often pay a higher energy bill for peripheral devices than for the CPU. Therefore, CPU-centric strategies, such as DVS-aware scheduling, must be reviewed to include external devices. Thus an active power manager must keep track of peripheral device usage and apply some heuristic to change their operating mode along the system lifetime. The decision of which devices will have their operating modes changed and when this will occur is mostly based on event counters maintained by the power management infrastructure, either in hardware or in software.

—As a matter of fact, critical real-time systems are almost always designed considering energy sources that are compatible with system demands. Power saving decisions, such as voltage scaling and device hibernation, are also made at design-time and thus are also taken in consideration while defining the energy budget necessary to sustain the system. At first sight, autonomous power management might even seem out of scope for critical systems. Nonetheless, complex, battery-operated, real-time embedded system, such as satellites, autonomous vehicles, and even sensor networks, are often modeled around a set of tasks the include both, critical and non-critical tasks. A power manager for one such embedded system must respect design-time decisions for critical parts while trying to optimize energy consumption by non-critical parts.

With these premises in mind, the next section briefly surveys the current scenario for power management in embedded systems, preparing the field for our proposal in the subsequent section.

4.1 Current Power Managers

Just like APIs and infrastructures, most of the currently available embedded system power managers focus on features exported by the underlying hardware. μCLINUX captures APM, ACPI or equivalent events to conduct mode transitions for the CPU and also for devices whose drivers explicitly registered to the power manager [Vaddagiri et al. 2004].

In TINYOS, OS-driven power management is implemented by the task scheduler, which makes use of the `StdControl` interface to start and stop components [Hill et al. 2000]. When the scheduler queue is empty, the main processor is put in *sleep* mode. In this way, new tasks will only be enqueued during the execution of an interrupt handler. This method yields good results for the main microcontroller, but leaves more aggressive methods, including starting and stopping peripheral components up to the application. When compared to μCLINUX, TINYOS delivers a lighter mechanism, more adequate to embedded systems, yet suffers from the same limitations with regard to usability and portability.

MANTIS uses an *idle* thread as entry point for the system's power management policies, which put the processor in *sleep* mode whenever there are no threads waiting to be executed [Abrach et al. 2003].

GRACE-OS is an energy-efficient operating system for mobile multimedia appli-

cations implemented on top of LINUX [Yuan 2004]. The system combines real-time scheduling and DVS techniques to dynamically control energy consumption. The scheduler configures the CPU speed for each task based on a probabilistic estimation of how many cycles they will need to complete their computations. Since the systems is targeted at soft real-time, multimedia applications, loosing deadlines due to estimation errors is tolerated. GRUB-PA follows the same guidelines, but addresses hard real-time requirements more consistently by imposing DVS configuration restrictions for this kind of task [Scordino and Lipari 2004].

Niu also proposed an strategy to minimize energy consumption in soft real-time systems through adjusts in the system's QoS level [Niu and Quan 2005]. In this proposal, tasks specify CPU QoS requirements through `(m,k)` pairs. These pairs are interpreted by the scheduler as execution constraints, so that a task must meet at least `m` deadlines for any `k` consecutive releases. The possibility to lose some deadlines enables the scheduler to explore DVS more efficiently at the cost of preventing its adoption in many (hard real-time) embedded systems.

Yet in the line of energy savings through adaptive scheduling and QoS, ODYSSEY takes the concept of soft real-time to the limit. The system periodically monitors energy consumption by applications in order to adjust the level of QoS. Whenever energy consumption is too high, the system decreases QoS by selecting lower performance and power consumption modes. In this way, system designers are able to specify a minimum lifetime for the system, which might be achieved by severely degrading performance [Flinn and Satyanarayanan 1999].

ECOS defines a *currency*, called *currentcy*, that applications use to *to pay for* system resources [Zeng et al. 2002]. The system distributes *currentcies* to tasks periodically accordingly to an equation that tracks the battery discharge rate as to ensure a minimum lifetime for the system. Applications are thus forced to adapt their execution pace according to their *currentcy* balances. This strategy has one major advantage over others discussed so far in this paper: the *currentcy* concept encompasses not only the energy spent by the CPU (to adjust DVS configuration), but the energy spent by the system as a whole, including all peripheral devices.

Harada explores the trade-off between QoS maximization and energy consumption minimization by allocating processor cycles and defining operating frequencies with QoS guarantees for two classes of tasks: real-time (mandatory) and best-effort (optional) [Harada et al. 2006]. The division of tasks in two parts, one *mandatory*, that must always be executed, and another *optional* that is only executed after ensuring that there are enough resources to execute the mandatory parts of all tasks is the basic premise behind *Imprecise Computation* [Liu et al. 1994], which is also one of the foundations of the power manager proposed in this work.

4.2 Proposed Power Manager

From the above discussion about currently available power managers for embedded system, one can conclude that no single manager consistently addresses all the points identified earlier in this section: leanness, real-time conformance, peripheral device control, and design-time decision awareness. We follow these premises and build on the API proposed in section 2 and on the infrastructure presented in section 3 to propose an effective autonomous power manager for real-time embedded systems.

For the envisioned scenarios of battery-operated, real-time, embedded systems, energy budgets would be defined at design-time based on critical tasks, while non-critical tasks would be executed on a best-effort policy, considering not only the availability of time, but also of energy. Along with the assumption that an autonomous power manager cannot interfere with the execution of hard real-time tasks (i.e., cannot compromise their deadlines), the separation of critical and non-critical tasks at design-time lead us to the following scheduling strategy:

—Hard real-time tasks are handled by the system as mandatory tasks, executed independently of the energy available at the moment. These tasks are scheduled according to traditional algorithms such as Earliest Deadline First (EDF) and Rate Monotonic (RM) [Liu and Layland 1973], either in their original shape or extended to support DVS.

—Best-effort tasks, periodic or not, are assigned lower priorities than hard real-time ones and thus are only executed if no hard real-time tasks are ready to run. Furthermore, the decision to dispatch a best-effort task must also take in consideration whether the remaining energy will be enough to schedule all hard-real time tasks.

—Whenever a best-effort task is prevented from executing due to energy limitations, a speculative power manager is activated in order to try to change components, including peripheral devices, to less energy-demanding operating modes, thus promoting energy savings.

With this strategy, the autonomous power manager will only be executed if energy consumption is detected excessive (i.e. a best-effort task has been denied execution) and time is available (i.e. a best-effort task would be executed). Non-interference between power manager and hard real-time tasks is ensured, in terms of scheduling, by having the power manager to run in preemptive mode, so that a hard real-time task would interrupt its execution as soon as it gets ready to run (e.g. after waiting for the next cycle).

This scheduling strategy has only small implications in terms of process management at the operating system level, but require a comprehensive power management infrastructure, like the one presented in section 3, in order to be implemented. In particular, battery monitoring services are needed to support the scheduling decisions around best-effort tasks and component dependency maps are needed to avoid power management decisions that could impact the execution of hard real-time tasks.

The operating mode transition networks introduced in section 3 as means to control the propagation of power management actions from high-level components down to the hardware can be used by the autonomous power manager to keep track of dependencies among components. Along with a list of currently active components maintained by the operating system, these transition networks build the basis on which peripheral control can be done by the power manager. For instance, if a task has an open file that is no longer being used, the power manager could track that component down to a flash memory and change its operating mode to standby.

Nevertheless, the compromise with real-time systems, requires our power manager to take API calls made by hard real-time tasks as “orders” instead of “hints”. We

assume that, if a hard real-time task calls the `power()` API method on a component to set its operating mode to *full*, then that component must be kept in that mode even if the collected statistics indicate that it is no longer being used and thus would be a good candidate to be shutdown. Otherwise, the corresponding task could miss its deadline due to the delay in reactivating that component.

5. IMPLEMENTATION IN EPOS: A CASE STUDY

In order to validate the power management strategy for embedded systems proposed in this paper, which includes an API specification, guidelines for power management infrastructure implementation through aspect-programs, and design constraints for the development of autonomous power management agents, the hypothetical *remote monitoring application* described in section 2 was implemented in EPOS and subsequently evaluated in terms of functionality and adequacy to the embedded system scenario.

5.1 EPOS Overview

EPOS, the Embedded Parallel Operating System, aims at building tailor-made execution platforms for specific applications [Fröhlich and Wanner 2008]. It follows the principles of *Application-driven Embedded System Design* [Fröhlich 2001] to engineer families of software and hardware components that can be automatically selected, configured, adapted, and arranged in a component framework according with the requirements of particular applications.

An application written based on EPOS published interfaces can be submitted to a tool that performs source code analysis to identify which components are needed to support the application and how these components are being deployed, thus building an execution scenario for the application. Alternatively, users can specify execution scenarios by hand or also review an automatically generated one. A build-up database, with component descriptions, dependencies, and composition rules, is subsequently accessed by the tool to proceed component selection and configuration, as well as software/hardware partitioning based on the availability of chosen components in each domain. If multiple components match the selection criteria, then a cost model is used, along with user specifications for non-functional properties, such as performance and energy consumption, to choose one of them[3].

After being chosen and configured, software components can still undergo application-specific adaptations while being plugged into a statically metaprogrammed framework that is subsequently compiled to yield a run-time support system. This application-specific system can assume several shapes, from simple libraries to operating system micro-kernels. On the hardware side, component selection and configuration yields an architecture description that can be either realized by discrete components (e.g. microcontrollers) or submitted to external tools for IP synthesis. An overview of the whole process can be seen in figure 15.

[3]Design-space exploration is currently being pursued in EPOS by making the cost model used by the building tool dynamic.

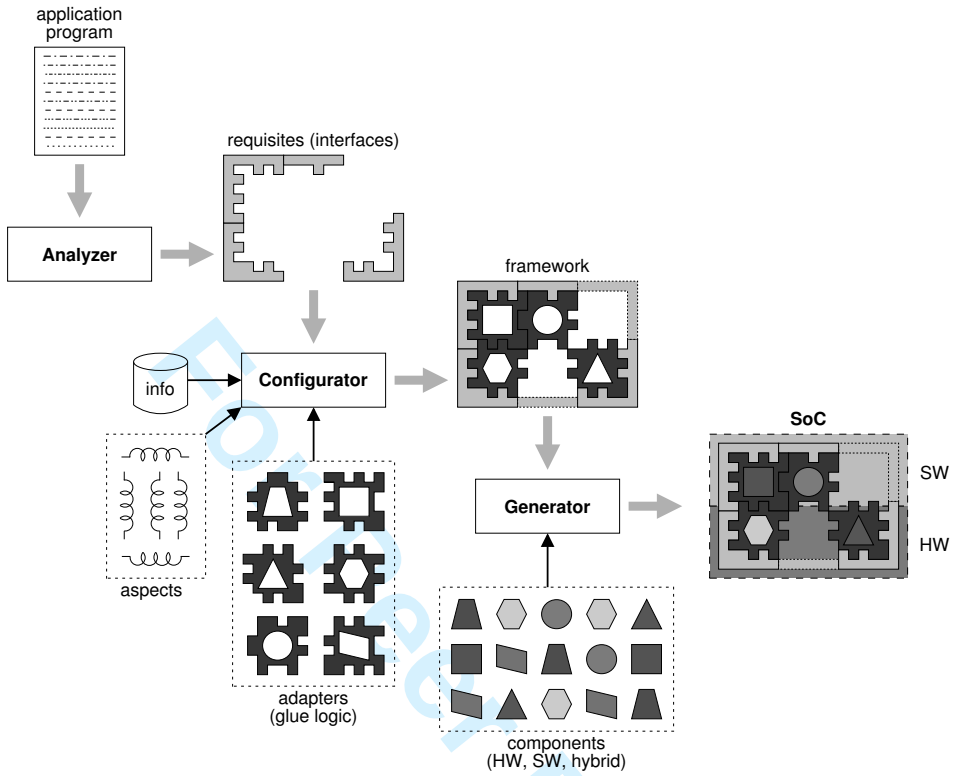

Fig. 15. Overview of tools involved in EPOS automatic generation.

5.2 EPOS Setup

The remote sensing application described in section 2 was implemented in EPOS through a C++ program very similar to the one shown in figure 10. The major difference lays in the inclusion of a fifth thread, `Calibrator`, that periodically calibrates the temperature sensor. The purpose of this addition is just to demonstrate the operation of the proposed autonomous power manager, so calibration is limited to calculate the average error after 100 samplings. The thread was created with `BEST_EFFORT` priority and a period of 100 seconds:

```
Periodic_Thread(&calibrator, BEST_EFFORT, 100000000);
```

When submitted to EPOS tools, the remote sensing program yielded a run-time library that realizes the required interfaces and a hardware description that could be matched by virtually any hardware platform in the system build-up database. We forced the selection of a well-known platform, the Mica2 sensor node [Hill et al. 2004] by manually binding `Communicator` to the CC1000 radio on the Mica2 platform and `Actuator` to a led.

The system is configured with a scheduling quantum of 15 ms. On each rescheduling, the scheduler checks if the next task is real-time or best-effort by assessing its priority. Best-effort tasks are only dispatched after confirming energy availability

Table III. Example monitoring system energy consumption under different configurations.

Configuration	1-Hour Energy (J)	365-Day Consumption (mAh)	Lifetime (days)
No Calib. (a)	5.07	4,112.29	479
RT Calib. (b)	8.13	6.596,81	299
PM Calib. (c)	6.66	5.400,00	365

with the power management infrastructure.

5.3 Measurements

In order to evaluate the remote monitoring system, we profiled its execution for time and energy with a digital oscilloscope. In the experiment, energy for the system was delivered by two high-performance alkaline AA batteries with a total capacity of 58320 J (5400 mAh at 3 V), in excess of table I estimates of what would be necessary to match the intended life-time of one year (i.e., 3580 mAh at 3 V).

The results of the experiment are summarized in table III, which presents the average energy consumption of the system for three different configurations: (a) executing without the `Calibrator` thread; (b) executing with `Calibrator` thread in hard real-time priority; (c) executing the `Calibrator` thread under the best-effort policy controlled by our autonomous power manager.

Each of the three configurations in the experiment was evaluated during approximately one week using new battery sets. From these measurements, we determined the average energy consumption per hour and extrapolated the total energy consumption for one year. We also estimated how long the system would be able to run on its energy budget under each configuration. Although battery discharge in this experiment is not linear, total consumption and lifetime estimates allows us to corroborate the benefits of the proposed active power manager. Decline in energy supply along time would induce the power manager to cancel even more recalibrations.

6. CONCLUSION

In this article, power management in embedded systems was addressed from energy-aware design to energy-efficient implementation, aiming at introducing a set of mechanisms specifically conceived for this scenario. A power management API defined at the level of user-visible system components was proposed and compared with traditional APIs. Its implementation was discussed in the context of the necessary infrastructure, including battery monitoring, accounting, auto-suspend and auto-resume mechanisms. An energy-event propagation mechanism based on *Petry Nets* was proposed and its implementation using *Aspect-Oriented Programming* techniques was depicted. The use of the proposed infrastructure by an autonomous power manager was also discussed, thus covering the main components of a modern power management system.

The proposed mechanisms were illustrated and evaluated through a didactic, yet realistic, example embedded system target at environment temperature monitoring. The example was described from early design stages down through a real implementation for the EPOS system on a Mica2 Mote, thus corroborating the proposed strategy and enabling readers to compare the associated mechanisms with other

proposals.

Therefore, the intended contribution of this article is not "yet another power manager for embedded systems", but the introduction of a broader and systematic way to deal with power management issues in embedded systems.

ACKNOWLEDGMENTS

I would like to thank and acknowledge former LISHA members Arliones S. Hoeller Jr, Geovani R. Wiedenhoft, Giovani Gracioli and Lucas Wanner for implementing many of the concepts and ideas presented in this article.

REFERENCES

Abrach, H., Bhatti, S., Carlson, J., Dai, H., Rose, J., Sheth, A., Shucker, B., Deng, J., and Han, R. 2003. Mantis: System support for multimodal networks of in-situ sensors. In *2nd ACM International Workshop on Wireless Sensor Networks and Applications*. 50 – 59.

Bacon, D. F., Cheng, P., and Rajan, V. T. 2003. A real-time garbage collector with low overhead and consistent utilization. *ACM SIGPLAN Notices 38,* 1, 285–298.

Bellosa, F. 2000. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of the 9th ACM SIGOPS European Workshop*.

Benini, L., Bogliolo, A., and Micheli, G. D. 1998. Dynamic power management of electronic systems. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*. ACM Press, New York, NY, USA, 696–702.

Chou, P. H., Liu, J., Li, D., and Bagherzadeh, N. 2002. Impacct: Methodology and tools for power-aware embedded systems. *DESIGN AUTOMATION FOR EMBEDDED SYSTEMS, Special Issue on Design Methodologies and Tools for Real-Time Embedded Systems 7,* 3 (Oct), 205–232.

Devadas, V. and Aydin, H. 2008. On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications. In *Proceedings of EMSOFT 08*. Atlanta, Georgia, USA, 99–108.

Flinn, J. and Satyanarayanan, M. 1999. Energy-aware adaptation for mobile applications. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*. ACM Press, New York, NY, USA, 48–63.

Fröhlich, A. A. 2001. *Application-Oriented Operating Systems*. GMD - Forschungszentrum Informationstechnik, Sankt Augustin.

Fröhlich, A. A. and Wanner, L. 2008. Operating system support for wireless sensor networks. *Journal of Computer Science 4,* 4, 272–281.

Harada, F., Ushio, T., and Nakamoto, Y. 2006. Power-aware resource allocation with fair qos guarantee. In *RTCSA '06: Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE Computer Society, Washington, DC, USA, 287–293.

Hill, J., Horton, M., Kling, R., and Krishnamurthy, L. 2004. The platforms enabling wireless sensor networks. *Communications of the ACM 47,* 6, 41–46.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. Cambridge, Massachusetts, United States, 93–104.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. 1997. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*. Lecture Notes in Computer Science, vol. 1241. Springer, Jyväskylä, Finland, 220–242.

Liu, C. L. and Layland, J. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20,* 1, 46–61.

Liu, J. W., Shih, W.-K., Lin, K.-J., Bettati, R., and Chung, J.-Y. 1994. Imprecise computations. *Proceedings of the IEEE 82,* 1 (Jan), 83–94.

Lohmann, D., Schröder-Preikschat, W., and Spinczyk, O. 2005. Functional and non-functional properties in a family of embedded operating systems. In *Proceedings of the Tenth IEEE International Workshop on Object-oriented Real-time Dependable Systems*. IEEEPress, Sedona, USA.

Monteiro, J., Devadas, S., Ashar, P., and Mauskar, A. 1996. Scheduling techniques to enable power management. In *Proceedings of the 33rd Annual Design Automation Conference*. Las Vegas, NV, USA, 349–352.

Niu, L. and Quan, G. 2005. A hybrid static/dynamic dvs scheduling for real-time systems with (m, k)-guarantee. *rtss 0*, 356–365.

Panasonic. 2004. *ERTJ Multilayer Chip NTC Thermistors Datasheet.* Panasonic.

Peterson, J. L. 1977. Petri nets. *ACM Comput. Surv. 9,* 3, 223–252.

Ren, Z., Krogh, B., and Marculescu, R. 2005. Hierarchical adaptive dynamic power management. *IEEE Transactions on Computers 54,* 4, 409–420.

Scordino, C. and Lipari, G. 2004. Using resource reservation techniques for power-aware scheduling. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*. ACM Press, New York, NY, USA, 16–25.

Tennenhouse, D. 2000. Proactive Computing. *Communications of the ACM 43,* 5 (May), 43–50.

Vaddagiri, S., Santhanam, A. K., Sukthankar, V., and Iyer, M. 2004. Power management in linux-based systems. *Linux Journal*.

Wiedenhoft, G. R., Junior, A. S. H., and Fröhlich, A. A. 2007. A Power Manager for Deeply Embedded Systems. In *12th IEEE International Conference on Emerging Technologies and Factory Automation*. Patras, Greece, 748–751.

Yuan, W. 2004. Grace-os: An energy-efficient mobile multimedia operating system. Ph.D. thesis, University of Illinois at Urbana-Champaign.

Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. 2002. Ecosystem: managing energy as a first class operating system resource. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. ACM Press, New York, NY, USA, 123–132.

Received November 2009; November 1993; accepted January 1996