# Peripheral State Persistence and Interrupt Management For Transiently Powered Systems

Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, Guillaume Salagnac

▶ **To cite this version:**

Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, Guillaume Salagnac. Peripheral State Persistence and Interrupt Management For Transiently Powered Systems. NVMW 2018 - 9th Annual Non-Volatile Memories Workshop, Mar 2018, San Diego, United States. pp.1-2. hal-01943919

**HAL Id: hal-01943919**
**https://hal.science/hal-01943919v1**

Submitted on 4 Dec 2018

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Peripheral State Persistence and Interrupt Management For Transiently Powered Systems

Gautier Berthou[1], Tristan Delizy[1], Kevin Marquet[1], Tanguy Risset[1], Guillaume Salagnac[1]
[1]Univ. Lyon, INSA-Lyon, Inria – Villeurbanne France

***Abstract*—Recently has emerged the concept of *transiently powered systems* powered by harvesting power and being able to retain information between power failures using non-volatile RAM. While existing solutions focus on purely computing systems, this article presents *Sytare*, a software layer designed to allow the use of non-trivial peripherals such as timers, serial interface or radio devices in transiently powered systems.**

***Index Terms*—NVRAM, internet of things, energy harvesting, sensors, ultra-low power, micro-architecture, compilers, instant on-off and transient computing systems**

## I. Introduction and problem statement

Using a battery in tiny embedded systems can be undesirable or even impractical [2]. In such cases, the system must harvest energy from its environment but it must then cope with an unreliable power supply. One obvious nuisance of *transient* power is that the system will lose every volatile state at each power failure. Recent advances in non-volatile memories allow to envision tiny systems that do not lose their data in case of power outage. However, naively replacing RAM with NVRAM has undesirable side-effects. To remedy this problem, recent works propose to detect when a power failure is about to happen, and then save processor state to NVRAM before halting execution. This mechanism is called checkpointing, and data structures that contain such data are called checkpoints. However, these studies tend to focus on the computational angle and ignore peripheral accesses.

The problem we address is to make hardware peripherals *persistent* across reboots so that the application does not notice power failures.

The first issue is to cope with the volatility of peripheral state. Capturing and restoring the internal state of peripherals require more complex techniques than doing so for application state. Existing works on transiently powered systems either ignore peripherals completely, or use hard-coded workarounds [3] to configure the hardware before restoring application state. We propose a technique to address this problem in the general case. Our approach is completely transparent for the application, and requires little modification of driver code.

The second issue is to cope with power failures occurring in the middle of a hardware request being serviced. Even if the state of a peripheral is non-volatile (either using non-volatile memory, or some software technique) a power failure may not be transparent for the user program. For instance, consider an application sending a radio packet using a `send_message()` function call. Now a power failure happens, in the middle of the transmission. At next boot, it would not make sense to "send half of the packet". Not only because the receiver may be gone, but also because the hardware has no concept of "half packet". This kind of problem arises even with simple peripherals such as an ADC (*Analog-to-digital converter*). If the power failure is to get unnoticed by application code, then the whole hardware access must be *retried*. We will refer to this issue as the *peripheral access atomicity problem*. The peripheral access atomicity problem has also been referred to as the *Broken Time Machine* problem [4].

The third issue is to deal with interrupts. When an interrupt signals that an event has occurred, the software stack must handle it, since interrupt handling may require to modify data and peripherals state shared with the (interrupted) application. Thus, interrupts must be handled carefully or they can lead the application to consistency issues.

Existing works do not address these three problems in a satisfactory fashion.

## II. The Sytare kernel

In this Section, we first consider that the execution flow can only be interrupted by powerloss detection interrupt, then we describe how we deal with other interrupts, finally the complete execution flow of Sytare is described.

### A. Sequential execution

Our approach to provide peripheral state persistence revolves around the interface between *application code* and *driver code*. We interpose a so-called *kernel* layer between the two, so as to intercept requests and responses. The Sytare *kernel* is responsible for persistence management, which includes saving and restoring application state to and from non-volatile memory. *Driver code* contains all functions which provide access to hardware features. In Sytare, if an application needs to invoke a driver function, it can only do so via *system call* (or *syscall*) mechanism implemented in the kernel. We used this denomination by analogy with the homonymous concept in classical kernels. In practice, a syscall is a thin wrapper around a driver function, adding the necessary features to address the atomicity problem.

The Sytare kernel ensures that, if the application is interrupted by a power failure, its state is saved and then at next boot it will restored. In the same manner, the Sytare kernel ensures that, if a syscall is interrupted by a power failure, then at next boot it will be re-invoked in the same conditions (arguments, hardware state, etc.). This piece of information is stored in non-volatile memory in a data structure we refer to as a *device context*. The device contexts are saved to persistent memory not upon power failures, but upon entering/exiting syscalls. Also, system calls are executed in a volatile fashion, i.e. nothing a syscall does is made persistent until execution returns to the application. For instance, we forbid application code to directly use memory-mapped registers to communicate with a hardware device. Instead, we require this service to be encapsulated in a driver function and invoked explicitly from the application. A driver may call primitives from other drivers, for instance our radio chip driver is built on top of the SPI driver, which itself requires digital I/O.

Restoring the state of a hardware device typically requires non-trivial operations like configuring some I/O pins, communicating over a serial bus (which itself should be initialized first), respecting certain timing constraints etc. While it may be conceivable for a persistence kernel to perform all these operations transparently, in Sytare we require some cooperation from the drivers developer: storing state in a *device context*, implementing a `restore()` function and a `save()` function for each driver.

A complete state machine of Sytare's control flow is given in the next Section.

### B. Interrupt handling

A standard approach for interrupt handling in operating systems consists in splitting interrupt handlers into two parts [1]. When an interrupt occurs, it is immediately handled by the operating system that executes a so-called *top-half* routine with IRQs disabled. This routine typically acknowledges the interrupt, and registers a deferrable *bottom-half* in charge of handling the lengthy operations associated with the interrupt. In Sytare, bottom halves are at the moment designed to be non-nestable, *i.e.*, no other bottom half can be run when a bottom half is interrupted, but interrupts are left enabled in order to be able to react upon powerloss occurrence.

While an interrupt top-half takes care of very low-level operations, the application developer is likely to request peripheral access from a bottom half. For instance if the application developer wants to read a radio packet upon radio reception interrupt, it might be natural to call relevant syscalls in the dedicated user-written interrupt handling procedures. To this extent bottom halves are allowed to use syscalls in the exact same way as application code would.

### C. Sytare's complete control flow

Figure 1 depicts as a state machine Sytare's control flow. Sytare always starts in *Boot* state, runs the corresponding code and switches to either *Init* state or *Restore* state depending on the presence of a former valid checkpoint. *Init* state initializes kernel variables and user application environment, then switches to *App* state. *Restore* state restores user context and device contexts and switches to a state depending on the powerloss detection that caused the checkpoint being restored happened respectively during the *App* state or the *Syscall* state. Whenever the application calls a syscall, control flow switches from *App* state to *Syscall* state. When a syscall returns, control flow switches back into *App* state. When powerloss detection occurs, control flow switches to *Checkpoint* state which performs the necessary memory transfers and waits for the hardware to shutdown.

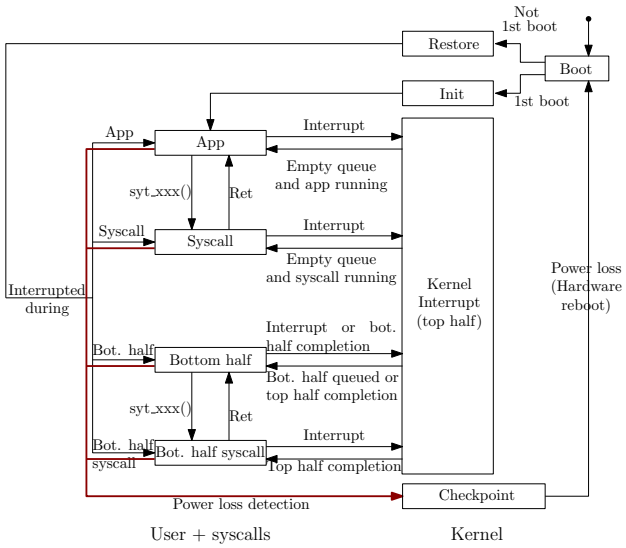


Fig. 1. Complete state machine of Sytare.

When an interrupt occurs in either state *App* or *Syscall*, the control flow transitions to state *Kernel Interrupt top half* in charge of performing interrupt acknowledgment and scheduling the corresponding bottom half. When a bottom half is scheduled, the control flow goes to state *Bottom half*. The state machine formed by the subset of states *Bottom half* and *Bottom half syscall* behaves exactly the same as the state machine formed by states *App* and *Syscall*.

When an interrupt other than powerloss detection occurs during *App* or *Syscall* state, code goes to state *Kernel Interrupt top half*. If the user has registered a bottom-half for the given interrupt, the top half schedules the bottom-half to be run before resuming the interrupted application or syscall. So when the top-half is done, code goes to state *Bottom half*. When the bottom-half is done, code goes back to *Kernel Interrupt top half* and then back to either *App* or *Syscall* depending on the state when the interrupt occurred. Interrupts are enabled in *Bottom half* and *Bottom half syscall*.

| Interrupt source | Top half execution time ($\mu$s) |
|---|---|
| GPIO | 20 |
| Radio packet reception | 1603 |

Fig. 2. Median top half execution times for common interrupt sources.

## III. Experimental results

We use a handful of benchmark applications with various levels of interaction with peripheral devices.

Basically, our experimental results show that:

- Sytare successfully saves and restores peripherals states even if powerloss happened during the execution of complex hardware services involving several drivers (radio frontend accessed through SPI bus).
- Sytare has low impact on performance: the execution time of application on top of Sytare is increased by 1-3% depending on applications. These results do not take into account the time spent saving/restoring contexts.
- The time spent to restore contexts because of powerlosses is heavily dependent on the devices used.
- Latency between interrupt occurrence and bottom half execution is rather small and acceptable even if the platform is powered only during a few milliseconds. Examples of latency, or top half execution time, are given in Figure 2.

## IV. Conclusion

We described Sytare, an operating system kernel that allows an application to execute without noticing power failures on a tiny embedded systems harvesting energy from its environment. We addressed especially the problem of making hardware peripherals *persistent* across reboots.

## References


[1] W. Dong, C. Chen, X. Liu, Y. Liu, J. Bu, and K. Zheng. "SenSpire OS: A Predictable, Flexible, and Efficient Operating System for Wireless Sensor Networks". In: *IEEE Trans. on Comp.* (2011).

[2] Hrishikesh Jayakumar, Kangwoo Lee, Woo Suk Lee, Arnab Raha, Younghyun Kim, and Vijay Raghunathan. "Powering the Internet of Things". In: *ISPLED'14: International Symposium on Low Power Electronics and Design*. 2014.

[3] B. Lucia and B. Ransford. "A simpler, safer programming and execution model for intermittent systems". In: *PLDI 2015: 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2015.

[4] B. Ransford and B. Lucia. "Nonvolatile Memory is a Broken Time Machine". In: *MSPC 2014: ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. 2014.