Peripheral State Persistence For Transiently Powered Systems

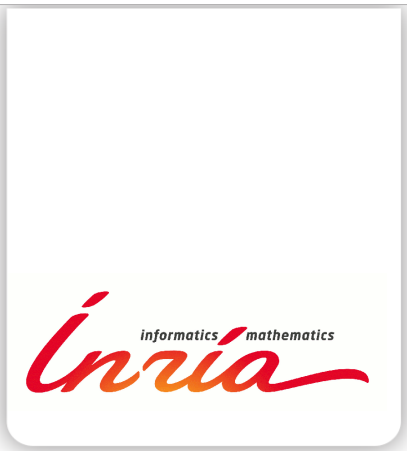
Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, Guillaume Salagnac

Peripheral State Persistence For Transiently Powered Systems

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

RESEARCH REPORT
N° 9018
February 2017
Project-Team Socrate

ISSN 0249-6399 ISRN INRIA/RR--9018--FR+ENG



Peripheral State Persistence For Transiently Powered Systems

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

Project-Team Socrate

Research Report n° 9018 — February 2017 — 42 pages

Abstract: Our society relies increasingly on digital technologies to communicate, seek medical information, travel, or have fun. These often-invisible technologies simplify our tasks and enrich our daily lives, while also developing the economy. Recently has emerged the concept of *transiently powered systems* powered by harvesting and being able to retain information between power failures using non-volatile RAM. This report presents a software layer called *sytare* that permits the use of non-trivial peripherals such as timers, serial interface or radio devices in transiently powered systems.

Key-words: Embedded Systems, NV-RAM, Energy Harvesting, Low-power, Wireless Sensor Networks, Internet of Things

* Univ Lyon, INSA Lyon, Inria, CITI, F-69621 Villeurbanne, France. e-mail: `firstname.lastname@inria.fr`

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Persitance de l'état des périphériques pour les systèmes alimentés de manière intermitente

Résumé : Notre société s'appuie de plus en plus sur les technologies numériques pour communiquer, consulter des informations médicales, voyager ou s'amuser. ces technologies simplifient nos tâches et enrichissent notre vie quotidienne, tout en développement de l'économie. Récemment a émergé le concept de *systèmes alimentés de manière intermitente*, alimenté en energie par harvesting et qui peuvent conserver l'information entre les coupures d'alimentation electrique en utilisant la RAM non volatile. Ce rapport présente une couche logicielle appelée *sytare* qui permet l'utilisation de périphériques non triviaux tels que les timers, les interfaces series ou les dispositifs radio dans les systèmes alimentés de manière intermitente.

Mots-clés : Systèmes embarqués, NV-RAM, Récolte d'énergie, Réseaux de capteurs de faible puissance, Internet des Objets

Contents

1 Background and Related Work **4**
1.1 Tiny Embedded Systems 4
1.2 Transiently Powered Systems 5
1.3 Non-volatile architectures 6
1.4 Program checkpointing 8
1.5 Discussion and problem statement 9
1.5.1 Problem statement 10

2 Contribution: the Sytare software **11**
2.1 Sytare structure 12
2.2 User program state persistence 13
2.3 Solution to peripheral state volatility 13
2.4 Solution to peripheral access atomicity 16
2.5 The checkpoint image structure 18
2.6 Simple Syscall and checkpointing operation 19
2.7 Complex Syscall and signaling example 20
2.8 Disscussion 23

3 Implementation **24**
3.1 Hardware platform prototype 24
3.2 Memory organisation 25
3.3 System boot 27
3.4 Implementation of the syscall wrappers 27
3.5 Device drivers 28

4 Evaluation **30**
4.1 Power supply 30
4.2 Metrics and variables 31
4.3 Benchmark applications 32
4.4 Benchmark application evaluation 32
4.4.1 Computational RSA application 33
4.4.2 Leds counter application 34
4.4.3 Sense and aggregate 34
4.4.4 Sense and send (WSN) application 35
4.4.5 Discussion 36
4.5 Kernel evaluation 36
4.5.1 System boot 37
4.5.2 Syscalls evaluation 38
4.5.3 Memory occupation 39

5 Conclusion and Perspectives **39**

1 Background and Related Work

This report presents a software layer developed in the context of an Inria support dedicated to the development of ultra low power devices. Transiently-powered systems (TPS) are a new class of batteryless computing devices that are of special interest today because the possible use of new non-volatile RAM technologie permits to foresee non-trivial applications for these devices. However, the arrival of these technologies imply an important shift in traditional low-level software stack: firmware and operating system. We present the first version of a software layer, called *Sytare*, that implements a generic methodology for using non-trivial peripheral (i.e. timers, ADC, SPI, Radio, LCD...) in transiently powered systems. Sytare could constitute the basic of future transiently powered devices operating systems.

1.1 Tiny Embedded Systems

Ubiquitous computing, also refered to as pervasive computing or ambient intelligence, is a paradigm where computing is not restricted to well-identified devices like computers or smartphones but embedded in virtually any object around us. With technology improving, this vision gradually becomes a reality as more and more "things" turn into "smart things". One typical example is that of smart cards that are widely used in many applications domains. But economic considerations prevent manufacturers from adding chips and software to anything: the benefits of the service must balance the additional cost. Thus ubiquitous computing platforms tend to be very resource-constrained microcontrollers. The architectures we are interested in in this work typically feature a slow processor (in the range of MHz), little memory space (a few kilobytes), and only a handful of peripheral devices.

These architectures form the low end of the "embedded systems" spectrum. Because they lack the computing resources of high-end platforms, they are not able to run a classical operating systems like *e.g.* Linux Microcontroller operating systems like FreeRTOS [GPPT16] or even Contiki [DGV04] typically require tens of kilobytes of memory and offer very limited functionality: interrupt processing, thread management, and sometimes a communication stack. Thus, many applications end up being programmed in a bare-metal fashion, with no OS support. The developer is directly in charge of managing the peripherals as well as all concurrent activities on the platform, typically in the form of interrupt service routines. The most common program structure of this kind of situation is the "super loop" [SG10] architecture, where the code roughly looks like the one represented in Figure 1. The main drawback of this programming model for our technological target is that it does not support unexpected power failure: the system can be stopped only when it decides to stop.

```
ISR deviceA_interrupt_routine()
{
   ...
}

ISR deviceB_interrupt_routine()
{
   ...
}

void main()
{
    hardware_init() ;

    __enable_interrupts();

    while(1)
    {
        task1_routine();
        task2_routine();
        ...

        __enter_low_power_mode(); // wait for interrupts
    }
}
```

Figure 1: Typical bare-metal programming style used for tiny embedded systems. There is no OS support, a *super loop* software structure is used instead (main inifinite loop + ISRs). This software structure does not support unexpected power failures.

The contribution of this report is to provide a new **checkpointing technique** for **transiently powered systems** that includes **non-volatile memory**.

1.2 Transiently Powered Systems

Embedded systems traditionally rely on battery power. This is true for high-end platforms like smartphones down to tiny nodes in a Wireless Sensor Network. The combination of battery capacity and average power draw determines the system operational lifetime: from a few days for a smartphone up to a few years for a Wireless Sensor Network. However there are also some situations where using a battery is undesirable or even impractical [JLL+14]. For instance, if the system is to be manufactured in large quantities (*e.g.* smart cards) then including a battery will significantly impact the unit cost. Also it would strongly increase the physical

size of the system, which might be unacceptable for the application scenario.

In such cases, the system must harvest energy from its environment and/or from external sources, *e.g.* solar power, piezoelectricity, thermal gradients, or electromagnetic fields [MZL+15]. One example is RFID systems, in which a (wired) *reader* transmits a radio-frequency signal strong enough to power nearby *tags* and have them send data back.

The last decade has seen a growing interest in designing such batteryless systems to be programmable with software. For example, Intel's Wireless Identification and Sensing Platform [BPS+08] is an attempt to bridge the gap between RFID systems and traditional sensor networks. Similar to a sensornet node, the WISP has a few sensors connected to a programmable microcontroller. Similar to a RFID tag, it has no battery and draws its power from the RF signal sent by a reader. More recently, researchers have presented arguments in favor of more and more miniaturization, and tackled the problem of miniaturizing the whole platform: the M^3 [LBL+13] is a $1.0mm^3$, general purpose, ultra low-power , configurable sensor node platform able to harvest energy from different sources, communicate wireless though. It embeds a Cortex-M0, few kilobytes of SRAM and few kilobytes of persistent SRAM.

One common characteristic of these systems is that they must cope with an unreliable power supply. Even when the energy source is active, the harvested power level is typically low [MZL+15] compared to what the system consumes in active mode. Storing energy in a capacitor is thus often necessary just to allow for useful work to be done in short bursts. For instance, conctactless smart cards must perform the *whole* transaction within a few hundreds of milliseconds, i.e. within the *lifecycle* of the device. If the transaction to be processed is longer, this is simply infeasible. Besides feasibility issues, the constraints imposed to the programmers of these devices are very tough as a very strong energy consumption prototyping must be done before releasing new software. Hence it is important to provide a new programming model that separates application layer and low level operating systems layer so as to implement non trivial application on transiently powered systems.

1.3 Non-volatile architectures

One obvious nuisance of transient power is that the system will lose every volatile state at each power failure. In a typical sensornet/TPS platform, this means losing the contents of CPU and RAM as well as all peripheral state.

Fortuntately in the last decade there have been significant advances in non-volatile memory (NVM) technology. Several NVM families are emerging which promise to blur and eventually remove the distinction between slow/non-volatile

"storage" and fast/volatile "memory" [BCGL11]. Examples include magnetoresistive RAM or phase change memory (PCM), among others. Current NVM technologies typically still suffer from slow write times, high write energy and/or limited write endurance [MSCT14]. It is nevertheless promising to use these technologies, but the place where they should be used within the memory hierarchy is not definitely decided today.

However in a transiently powered system, naively replacing RAM with NVRAM has undesirable side-effects. Because power failures are frequent, they can occur in the middle of a (non-volatile) data structure being modified. When the platform reboots, the program will restart with inconsistent data [RL14].

To remedy this problem, Jayakumar et. al [JRR14] propose to detect when a power failure is about to happen, and then save processor state to FRAM before halting execution. When the platform reboots, reloading processor state enables the program to resume from where it was interrupted.

A more direct approach would be to change the architecture and make the processor itself non-volatile [LLL+15]. Indeed it is tempting to implement every single storage element (*e.g.* flip-flops) in the platform with NVRAM. Because of the performance limitations of NVRAM, most non-volatile architectures adopt a hybrid approach : each traditional register is not replaced but complemented with a non-volatile counterpart [MZL+15]. The result is called a *non-volatile flip-flop*. In addition to the usual *read* and *write* operations, it is able to *save* and *restore* its contents to and from non-volatile storage. By triggering these operations at the right time, the system can appear as non-volatile without having to pay for the performance hit of NVRAM.

For instance Bartling et al. [BKC+13] design and fabricate such a non-volatile microcontroller. Their platform has 10 kB of ROM, 8 kB of SRAM, and 64 kB of Ferroelectric RAM (FRAM). Upon detecting a power failure, the chip automatically saves all CPU and peripheral registers to FRAM (∼320 bytes of data).

This kind of approach is interesting in terms of architecture but has a major limitation in terms of software programming. On the one hand, storing a program data structure in NVRAM makes it persistent, but also means that each access will be slow and/or energy-expensive. On the other hand, storing data in RAM gives good execution performance, but brings back the problem of volatility. For this reason, most non-volatile architectures actually employ a combination [BKC+13] of both RAM and NVRAM, such as for instance the The MSP-EXP430FR5739 board from Texas Instrument including 15kB of FRAM together with traditional RAM. The problem of power failures and persistence is left for the programmer to cope with. Indeed, backing every bit of RAM with a non-volatile flip-flop would not be feasible. Not only would it be very ineffective in terms of silicon area, but also the *save* operation would incur a massive current spike because of the high

write energy of NVRAM [LLL+15].

1.4 Program checkpointing

In the previous section, we discussed how hardware techniques alone are not well suited to achieve system persistence. In this section, we review software techniques with a similar objective and give details on these mechanisms, allowing to position the contributions presented in the remainder of this paper. The first paper on checkpointing for small embedded systems [ODV+09] is not about power failures but about debugging. In the context of sensornet testbeds, the authors design a mechanism for capturing the state of each node in the network. The user can then load this state image in a platform emulator (*e.g.* to inspect program state for debugging purposes) or inject a set of such images in all nodes of the network (*e.g.* to replay a past experiment with high fidelity). The checkpointing operation is triggered by a command received on the serial port. The capture (resp loading) mechanism itself is implemented within the serial port interrupt handler. The state images themselves are never stored on the node, but streamed to/from a PC host via serial.

Another influential paper is Mementos by Ransford et al. [RSF11]. The target of this work is the WISP "computational RFID" platform [BPS+08]. Because of the unstable nature of RF power harvesting, the node can run out of energy at any time. To prevent state loss, Mementos periodically interrupts the application and measures the remaining energy level. When above a user-specified threshold, the application is resumed. Otherwise Mementos saves the CPU registers and the contents of RAM to flash memory. This technique does make power failures transparent to the application program. However Mementos is designed specifically for flash, which leads to unsatisfactory performance. Because of flash's high write energy and slow write time, each checkpoint save operation is very expensive. Also, a region of flash memory cannot be overwritten and must be erased first. This means that our previous checkpoint cannot be updated in-place but must be re-captured entirely every time. The runtime overhead induced by Mementos is thus quite high, sometimes taking up the majority of execution time. Still this approach is promising and several studies have since explored similar techniques.

An obvious enhancement of this technique is to target a different non-volatile memory technology. With Hibernus [BWM+15] Balsamo et al. propose to save program state to FRAM, which yields significant time and energy savings compared to Mementos. Also, instead of interrupting the application on a regular basis, they use a hardware device to trigger the checkpointing operation only once when the power is about to be lost. The experimental results of Hibernus confirm that using RAM and NVRAM side by side and using checkpoints to compensate for power failures is a viable strategy.

But using addressable NVRAM also makes it possible to design a more sophisticated checkpoint data structure. To reduce the amount of NVRAM writes as much as possible, Ait-Aoudia et al. [AAMS14] propose an incremental checkpointing scheme. The idea is to view memory as divided in fixed-size blocks, and to only save the blocks which have been changed since the last checkpoint to date. If the application has a large RAM footprint, this approach can significantly reduce the amount of data written to NVRAM at each lifecycle.

Bhatti and Mottola [BM16] take this idea one step further. Instead of copying memory contents as opaque data, they distinguish between *stack*, *globals*, and *heap* regions. Because each region has a particular internal structure, saving it entirely is suboptimal. For instance, copying the "empty space" above the top of stack is pointless and may be safely avoided. Another example is the heap region, where each object can be copied individually rather than including empty space in between. The authors devise several such checkpointing schemes with increasing degrees of sophistication. Then they evaluate checkpointing performance against various benchmark programs and show that no single scheme performs best.

1.5 Discussion and problem statement

Bell's Laws states that a new class of less expensive computers is developed approximately every decade by using fewer components than state-of-the-art computing system [Bel08]. Transiently Powered Systems are a new class of tiny communicating systems with no battery and little computing resources. Their small form factor allow them to be embedded everywhere. However, harvesting energy in the environment is unreliable and the system will frequently run out of power, typically several times per second. This makes application programming a difficult task for the software developer.

As discussed in the previous sections, various techniques have been proposed to mitigate this problem and allows the program to make progress despite/across reboots. However, these studies tend to focus on the computational angle and ignore peripheral accesses altogether. Yet by definition no embedded program is purely computational. If we want power failures to be transparent for the application, then we must ensure that both program state and peripheral state persist across reboots.

Simple access peripherals. If the state of a hardware peripheral is perfectly mirrored in its interface registers, and if these registers are directly addressable from the processor, then it is enough to save and restore their value just like any other data structure. We will refer to these "simple access" peripherals as *type 1*. Most papers referenced in the previous section happen to only use type 1 peripherals.

This type of peripheral is very limited, one could include leds and button of GPIOs in it. But even on a small embedded system, the vast majority of the peripherals require more sophisticated mechanisms.

Constrained access peripherals. For instance, a peripheral may impose a certain protocol on the order its registers must be accessed: the configuration registers are read-only most of the time, and must be unlocked by writing a magic value into one particular address. Only then the other registers can be configured, and finally the program must lock the peripheral again by writing another magic value. No "generic" checkpointing mechanism will work correctly for this peripheral. Another instance of access protocol is timing constraints: some peripherals have intrinsic delays in their initialization sequence. If the program doesn't respect these delays, then the device may end up incorrectly configured. Depending on the situation, we either have to wait a for fixed duration, or we have to poll a certain signal telling us when to proceed. We will refer to such "constrained access" peripherals as *type 2*. They include most traditional peripherals: timers, ADC, serial ports, etc.

Indirect access peripherals. A third class of peripherals is not even addressable from the processor, but must be accessed through another peripheral. For instance, on a typical sensornet platform, the radio transciever would be connected to the microcontroller via a serial bus. In that case, we have to restore the serial port controller first, and then use it to talk to the radio itself. In addition to the precedence constraint itself, accessing each configuration register of the radio requires a serial communication. For this kind of peripheral, saving and restoring state involves a lot more work than just memory accesses. We will refer to such "indirect access" peripherals as *type 3*.

1.5.1 Problem statement

The problem we address in this paper is: how to make hardware peripherals *persistent* accross reboots so that the application doesn't notice power failures. This problem holds two aspects, *state volatility* and and *access atomicity*.

Peripheral state volatility problem The first issue is how to cope with the volatility of peripheral state. As discussed above, 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 [LR15] to configure the hardware before restoring application state. In this paper, we propose a technique

to address this problem in the general case. Our approach is completely transparent for the application, and requires little modification to driver code.

At first sight, one could argue that embedding non-volatile memory in every component of the platform would be enough to solve this problem. However, non-volatility in itself is not sufficient to guarantee the correct behaviour of the system.

Peripheral access atomicity problem The second issue is how to cope with power failures occuring 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 a scenario where the application wants to send a radio packet using some `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 the second half of the packet". Not only because the receiver may be gone, but also because the hardware has no concept of "half a packet". This problem may happen as soon as a hardware access cannot be simply resumed after a shortage. This is the case even for surprisingly simple peripherals such as an ADC (*Analog-to-digital converter*). In these cases, if we want the power failure to go unnoticed by application code, then the whole hardware access must be *retried*. We will refer to this issue as the *peripheral access atomicity problem*.

Existing works don't address these two problems in a satisfactory fashion. For instance, Dino [LR15] requires the programmer to manually insert chekpoint barriers in their code and guarantees that execution will resume at one of those points after a reboot. This approach somehow solves the atomicity problem, but does nothing about the state volatility problem. Thus, it doesn't allow the system to use complex peripherals as an active RF chip. The peripheral access atomicity problem has also been referred to as the *Broken Time Machine* problem [RL14]. In the next section we present our approach to tackle both aspects of the *peripheral state persistence problem*.

2 Contribution: the Sytare software

Our approach to provide peripheral state persistence revolves around the interface between *application code* and the *driver code*. The idea is to interpose a so-called *kernel code* layer between the two, so as to intercept requests and responses. This enables the kernel to know whether the system is executing application code or accessing the hardware, which will be useful to solve the atomicity problem. Before returning to the application, the kernel captures the state of the driver and thus its

underlying device. This information is stored in NV memory in a data structure we refer to as a *device context*.

When a power failure happens, the kernel saves a copy of *application state* from RAM to NVRAM, along with all device contexts. When power comes back on it loads these device contexts back to memory, and invokes the *restore()* primitive of each driver. This primitive is responsible for reading in the device context and bringing back the hardware in the required state. Then the kernel restores application state to RAM and execution can resume transparently.

However this scenario is only valid if the power failure occurs while executing application code. If a power failure occurs while executing driver code, then at next boot it will not make sense to *resume* execution. Instead, the hardware request should be *retried* from the beginning. We use the term *system call* to describe a function call from application code to driver code. 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) To that end, 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.

In this section we discuss all these mechanisms in more detail, and argue that they produce the correct behaviour for all three types of peripherals described in section 1.5.

2.1 Sytare structure

In bare-metal embedded software, application logic and low-level driver code may or may not be clearly separated. In Sytare, we draw a distinction between three possible kinds of code: application, driver, and kernel.

As expected, *application code* encompasses higher-level functions as well as library code. As such, *application state* is composed of all the global variables of these modules, as well as the contents of the execution stack (local variables, control flow) and CPU registers.

The Sytare *kernel* is responsible for persistence management, which includes saving and restoring application state to and from non-volatile memory.

We define *driver code* as being all functions which provide access to hardware features. 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. This is implemented with ordinary function calls.

However, we require the application to invoke a driver function only via the *system call* mechanism, implemented in the kernel. In practice, a syscall is a thin wrapper around a driver function, adding the necessary features to address the atomicity problem.

2.2 User program state persistence

To achieve persistence of the user program state, Sytare implements a checkpointing mechanism. A hardware device detects an imminent power failure (cf section 3.1) and interrupts the application. The kernel then copies application state to a non-volatile data structure we refer to a *checkpoint image*. At the beginning of the next lifecycle, the kernel loads this image back to RAM and execution resumes.

Like other similar operating systems [RSF11; AAMS14] Sytare maintains two checkpoint images at all times for resilience. The idea is to keep the last valid image intact while the next image is being built. This way, even if a crash (*e.g.* power failure) occurs during the checkpointing operation, the system will be able to recover at next boot.

Ensuring consistency between peripheral state and program state

The kernel must ensure consistency between peripheral persistent data and the program checkpoint done at the power loss to avoid undefined behaviour. This issue is addressed in Sytare by duplicating the driver data persisted in NVRAM with the same strategy than for program checkpoints. The kernel will build the peripheral checkpoint over time at the end of each syscall from the restored values and when a power loss occurs it will complete this checkpoint with the user program executional state. If the kernel lacks time to checkpoint entirely the program state, the kernel will reboot from the last complete checkpoint (peripherals + program state), wasting the last lifecycle to ensure consistency. This case can be encountered when the power loss occurs during system operations, for example committing a peripheral driver device context, or if the platform is consuming more than usual just before power loss *e.g.* sending a message via RF. This approach ensures the reliability of the system potentially facing power loss at any moment as a TPS to be still able to continue its tasks.

2.3 Solution to peripheral state volatility

As we discussed in section 1.5, checkpointing memory contents is not enough when the system includes hardware peripherals. 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*, and implementing a `restore()` function.

The `device_context_t` data structure In general, the state of the hardware device is somehow reflected in program variables in the driver. For instance, a LED driver will typically have one boolean flag for each diode, to remember whether the diode is on or off. If the device has a more complex state space, *e.g.* a finite state machine with several control modes, then the driver will keep track of the current mode with a variable, and so on.

Rather than using global variables within the driver, in Sytare we require all this information to be explicitly encapsulated inside a so-called *device context* data structure. This change is illustrated in Figure 2 on a simple example. Thanks to these structures, the Sytare kernel can take a snapshot of the state of each device at various points in time. In addition to the application state, a checkpoint image contains one device context for every hardware peripheral as it will be discussed further.

```
char led_state[LED_COUNT];

void led_switch_on(int led_number)
{
  HW_REGISTER_FOR_THIS_LED=1;

  led_state[led_number]=1;
}
```

```
typedef struct {
  char led_state[LED_COUNT];
} led_context_t ;

led_context_t *led_context;

void led_switch_on(int led_number)
{
 HW_REGISTER_FOR_THIS_LED=1;

 led_context->led_state[led_number]=1;
}
```

Figure 2: Illustration of the `device_context_t` data structure. On the left is some typical driver primitive from a bare-metal application. On the right, the same function has be modified to comply with the Sytare kernel. Note that only the type is defined in the driver code. The actual device context instances are allocated and managed by the kernel.

The `restore()` function Any device driver typically offers some `init()` primitive which performs the correct initialization procedure. This would be invoked for instance at the beginning of the `main()` function, before entering the main infinite loop.

Sytare requires each driver to provide an additionnal `restore()` primitive which will be called upon restoring a checkpoint. At boot, the kernel restores all device

```
void foo_restore(void)
{
    // hardware initialization
    foo_init();

    // restore configuration parameters
    foo_configure(foo_device_context->settings);

    // restore control mode (for FSM drivers)
    switch(foo_device_context->control_mode) {
        case DRV_MODE_0 :
            // bring back hardware in "DRV_MODE_0"
            foo_switch_to_mode_0();
            break;
        case DRV_MODE_1 :
            // bring back hardware in "DRV_MODE_1"
            foo_switch_to_mode_1();
            break;
        ...
        default :
            // unknown mode: can't happen
            kernel_panic();
    }
    return;
}
```

Figure 3: Illustration of the `restore()` primitive for a hypothetical driver. This listing illustrates quite a complex scenario, with both configuration parameters to be configured and some control mode to be restored. In most practical cases the restore function will only involve a subset of these operations.

contexts to memory, and then invokes the `restore()` function of each driver in succession. As illustrated by Figure 2.3, this function is responsible for initializing the hardware and then bringing it back to the required state as described by the device context. In the simplest scenario, this operation may consist in simple call to `init()`. However for more complex peripherals which require indirect access and/or impose certain access constraints, the `restore()` function will be more complex accordingly. Still it only makes use of existing features already present in the code, so in most cases adding such a function to an existing driver should be straightforward for the developer.

We noted earlier that some driver A may use the services of some other driver B in the system. Thus the kernel must ensure that driver A's `restore()` function is called only after B has been restored, etc. In the general case, this would translate

to a dependency graph where nodes are drivers and edges are precedence constraints. At each boot the kernel would need to compute a valid evaluation order and restore drivers in that order. However in our implementation we sidestepped this issue by ordering the drivers manually.

2.4 Solution to peripheral access atomicity

In a Transiently Powered System, application code and driver code are not to be treated equivalently when power failures occur. Application code function can be interrupted at any point by the checkpointing procedure. At next boot it will be restored transparently and resume execution exactly where it left. However if a driver operation is interrupted, then it must be considered entirely failed as the kernel can't ensure state consistency at lower granularity, as we discussed in Section 1.5. We have denoted this issue as the peripheral access atomicity problem, our solution makes use of a so-called *system call* layer that will ensure peripheral access atomicity.

System call To solve this problem we draw a distinction betweem two types of function calls. Within the application, or within driver code, ordinary function calls happen as usual. However the application may only invoke a driver function through a well-identified kernel interface. We denote this mechanism a *system call*, by analogy with the homonymous concept in classical kernels.

The contract between the application and the Sytare kernel is that a syscall will be executed entirely within one lifetime. If a power failure happens in the middle of servicing a syscall, then at next boot the OS will transparently *retry* the call instead of just resuming it.

Ensuring system call atomicity In cases where a power failure happens during the execution of a driver function, we want to ensure that at next boot it will be re-executed from the beginning. Thus we prevent anything belonging to the drivers from being saved by the checkpointing operation described in Section 2.2. If our target platforms offered hardware support for memory protection, we could place the application in one protection domain and drivers in another, like is done in classical kernels. However in tiny microcontrollers there is typically no such support, so we have to rely on software mechanisms.

In Sytare we isolate the application from the "driver land" by clearly separating their respective memory regions. For global variables and data structures, we have seen in Section 2.3 how each driver is explicitly given a *device context*. These are allocated separately from the application state, and considered differently for inclusion in the checkpoint image. To isolate local variables as well as the control

flow, we switch to a separate execution stack for executing driver calls. This so-called *OS stack* is never included in the checkpoint image, which guarantees that any partial progress inside a driver is volatile. The stack switch happens in the syscall wrapper invoked by the application. The wrapper is responsible for switching stacks and for forwarding the function arguments to the actual driver function. Also, it saves a copy of these arguments in the checkpoint image together with the syscall number.

In case the system call is interrupted by a power failure, then the checkpointing operation will be triggered, but it will only have to save application state. All information required to retry the call is already saved to non-volatile memory. At the next boot, the kernel restores application state to memory and checks whether a syscall has been interrupted. In that case, the syscall arguments are repopulated from the checkpoint image and the syscall is invoked afresh. Otherwise execution resumes directly to application code as usual.

Of course upon successfully returning from a syscall, the kernel wrapper erases the saved syscall number and arguments from the checkpoint image and switches back execution to the user stack.

Restoring the correct peripheral state before retrying a syscall When the system boots, the kernel reads the checkpoint image which contains application state to be reloaded as well as all device contexts to be restored. But so far we have not discussed how and when the device context are saved into the checkpoint image. In this section, we argue that they must be saved when returning from a system call. Two distinct boot scenarios must be considered: resuming app execution, and retrying a syscall.

In the first case, we want to restore each peripheral to the same state it was in just before the power failure. Thus, the saved device context has to describe the peripheral at that point in time, in the previous lifecycle. Going further back in time, we observe that a device context can only change as a result of executing driver code. After the last syscall returns, all device contexts will be left untouched, so it is safe to save them upon returning from the syscall.

In the second scenario, we want to restore each peripheral to the same state it was just before executing the interrupted syscall. Like we did above, let's consider the previous lifecycle and go back in time, starting from the beginning of the interrupted syscall. Here as well, we observe that the last time any device context was modified is during the preceding syscall.

To accommodate both cases with a single mechanism, we choose to save a copy of all device contexts upon returning from a syscall back to application code. This yields correct behaviour in the first scenario, even if saving the contexts later would work as well. In the second scenario, this approach ensures that all incomplete

hardware operations will be forgotten when the power failure happens.

Optimization: selective persistence of device contexts Obviously, saving a copy of all device contexts to NVRAM each time a driver call returns would imply a significant performance cost. To avoid this penalty, we introduce a selective persistence mechanism. The idea is to observe that each syscall will likely touch only one or two device contexts, and as such it is useless to save them all. Also, if a syscall does not change the state of the underlying device, or brings it back to the same state before returning to the application, then it is useless to save anything at all.

To support this scenario, we introduce a notification mechanism in the form of a `signal(driver_id)` kernel primitive. Each driver must call this primitive when its device context has changed. The kernel just marks the corresponding entry as "dirty" (i.e. modified) but does nothing else just yet. If the syscall involves serveral device drivers calling each other, then any number of device contexts may be signalled as modified. When the syscall ends successfully, the kernel wrapper does a `commit()` operation, i.e. it brings the checkpoint image up to date with all the modified device contexts, before returning to the application.

All these updates are done in the *"next"* checkpoint image, which is built piece by piece throughout the lifetime. The *"previous"* checkpoint image is left intact, so that we can always resume from there if something goes wrong. To support the case where no syscall happens during a lifetime, or if all syscalls are read-only operations, the kernel initializes (at boot time) the "next" checkpoint image with a copy of all device contexts from the "previous" image. In some cases, the "next" checkpoint image will be left intact for the entire lifetime, only to be made complete with a copy of application state when a power failure is detected.

Due to implementation specificities, the current prototype limits the arguments of a syscall to four. Our system doesn't rely on having a volatile memory zone for the work variables of the kernel, as we could run only on non-volatile memory.

2.5 The checkpoint image structure

The kernel, when power comes back, will restore a *complete checkpoint image* to resume the user application execution. This checkpoint image contains information describing the application state, the kernel state and the drivers state as illustrated in figure 4. To ensure that the system will be able to resume execution even if the checkpointing fails, the kernel maintains two checkpoint images:

- `last` checkpoint image: contains the last complete checkpoint image, if the next checkpoint is not correctly completed, this image will be resumed as fallback.

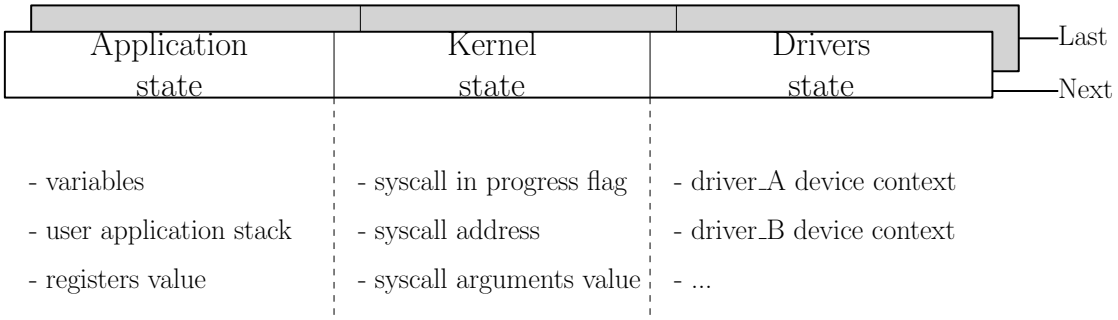


Figure 4: Illustration of the system checkpoint image content.

- `next` checkpoint image: contains the last committed state of the drivers device contexts and syscall status but no application state during the system lifecycle. This part will be checkpointed when power loss occurs. This checkpoint image is initialized at boot time with the resumed state of the drivers device context and is incrementally built.

2.6 Simple Syscall and checkpointing operation

In the scenario described by figure 5, emphasis was put on both showing the internal mechanisms of peripheral persistence and checkpoint consistency as implemented in Sytare. Initially, the application state is *App_0* and peripheral "A" has state *A_0*. The user application requests access to peripheral "A", which has to be done through Sytare API. Between the beginning of the represented user code section and the call to kernel function `syt_drvA_fn()`, the state of the application has changed from *App_0* to *App_1*, as every instruction impacts the volatile state of the system. At this point, peripheral "A" and its associated driver are still in state *A_0*.

Syscall sequence: Function `syt_drvA_fn()` calls its associated driver function, named `drvA_fn()` as `syt_drv<...>()` functions are wrapper functions that refer to `drv<...>()` functions. When the driver function is done with changing the state of the peripheral, the new state itself, referred to as *A_1*, is naturally recorded in volatile memory. Thus it needs to be recorded into the "Next" image for persistence purposes. In order to notify the kernel that peripheral "A" had its state changed, the driver calls `syt_signal()` primitive. Then the driver returns to the calling kernel function which performs a commit operation using `syt_commit()` primitive. The commit operation persists the new state of the peripheral into the "Next" image. Then the kernel wrapper function returns to the calling user code, which is still in state *App_1* given the fact that no user code was executed between the

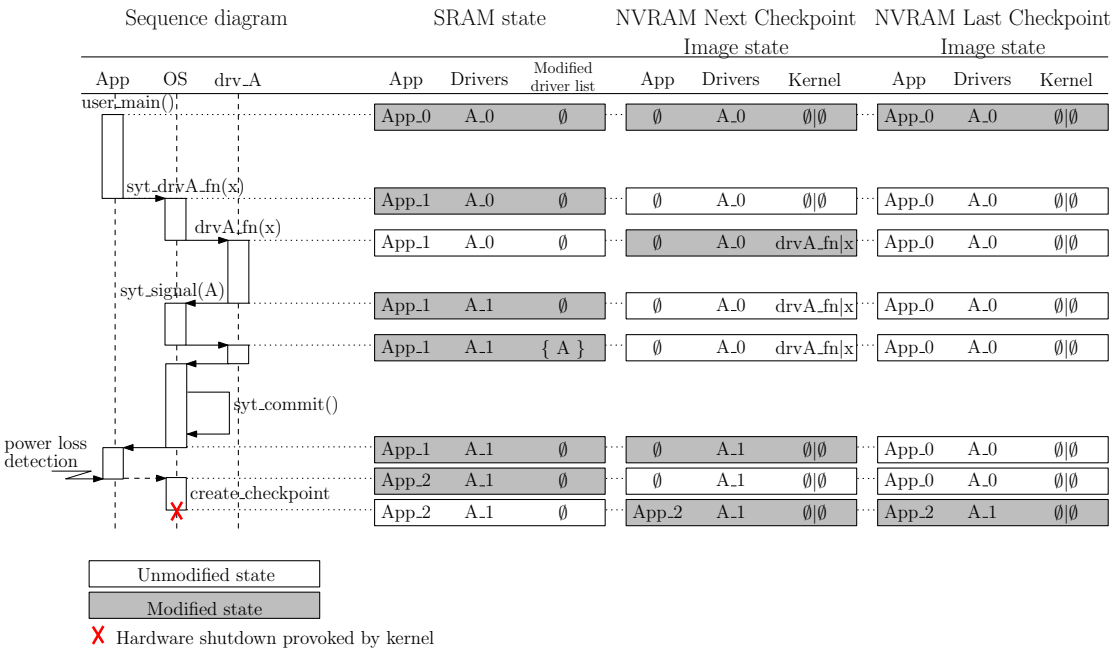


Figure 5: Sequence diagram of a simple Syscall with SRAM and NVRAM kernel data structures content. A power loss is detected while running user application after the syscall returned to application.

moment when the kernel started to work and the moment the kernel returned to user application.

Checkpointing operation: User code is resumed and runs a certain amount of instructions, changing application state from *App_1* to *App_2*. Simultaneously a power loss is detected and handled by a kernel interrupt routine. The interrupt routine persists the application state into the "Next" image since application code can only affect application state directly. Finally the kernel makes the "Last" image pointer point to the "Next" image. Now the system is ready to shutdown and it is ensured that application state *App_2* and driver state *A_1* will be restored on the next boot, *i.e.* the environment will be set up for the application to resume properly.

2.7 Complex Syscall and signaling example

In the figure 6 we detail the sequence of calls and data structure modification resulting from a syscall done in the user application. Recall that the Sytare driver mechanism is added to a classical checkpointing which is triggered when the power is lost, we did not represent the power loss in figure 6 hence the application state

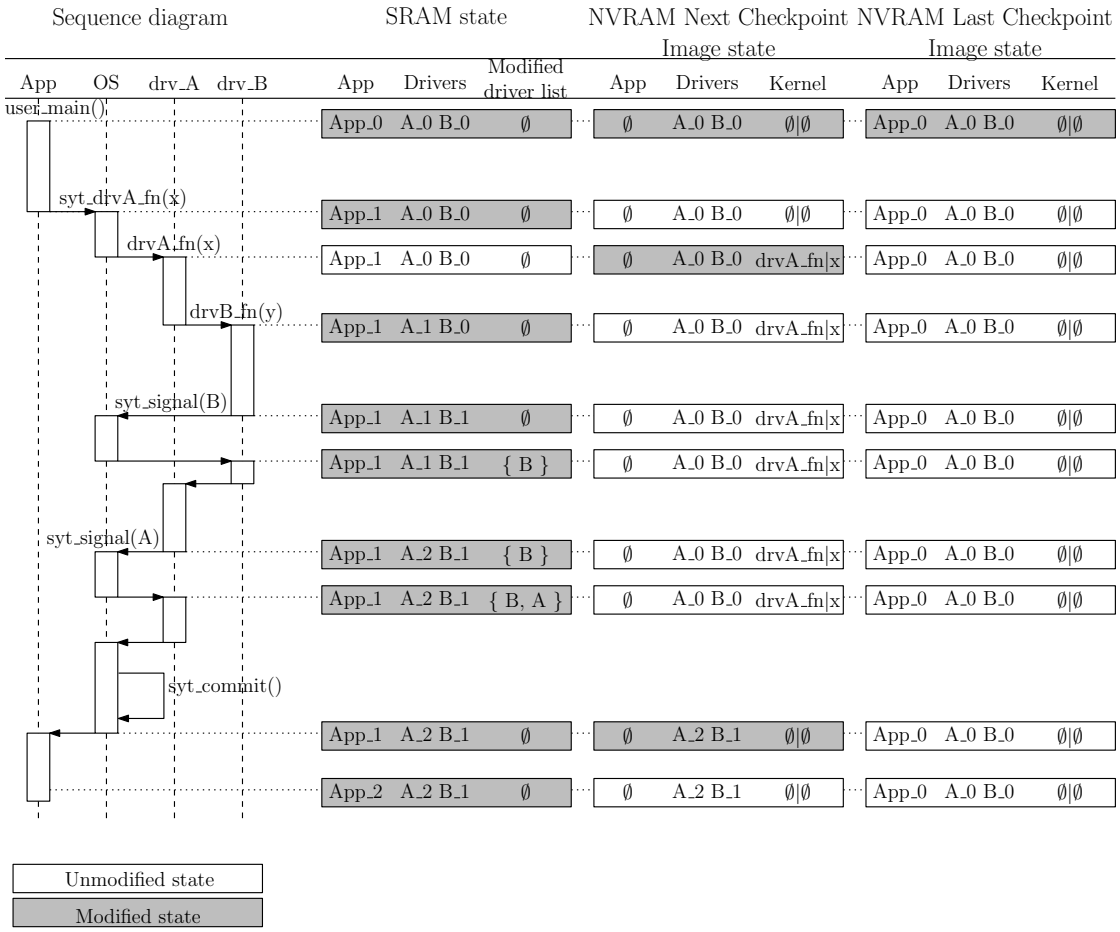


Figure 6: Sequence diagram of a complex Syscall with SRAM and NVRAM kernel data structures content and two (nested) drivers calls.

is not saved in NV-RAM on this sequence. The aim of the application developer is here to call the driver's function `drvA_fn()` with parameter `x`, accessing the underlying hardware peripheral "A". In order to illustrate the inter-driver call we make the supposition that in order to access peripheral "A" the system has to access peripheral "B" hardware. The diagram describes the function call sequence passing from the application to the kernel layer before executing some driver code. We describe accordingly the modifications done in the system different memory structures.

The "SRAM state" describes the volatile kernel state containing the application currently executed, the RAM located drivers device contexts and a list of the modified drivers device contexts compared to the last non volatile image persisted by the kernel. The "NVRAM Next Checkpoint Image state" and "NVRAM Last Checkpoint Image state" describe the images built by the kernel in non volatile memory from whitch a checkpoint can be restored. The "Last" image is the state restored at kernel boot `i.e.` the start of the sequence diagram time line. The "Next" image is the checkpoint image the kernel builds during its execution to survive the future power loss. So it does not contain any application state description as the power loss did not yet occur but incrementally builds the device context image waiting for the power loss to occur before completing the checkpoint image with the application state.

The syscall can end up in three different scenarios described among the following: it can finish without power loss, a power loss can occur and the kernel successefully checkpoints the application state or when the power loss occurs the kernel fails to checkpoint the application state.

Normal execution: This is the case represented in the diagram. Instead of calling directly the driver's function the user application uses the kernel wrapper associated to the driver. The kernel then starts by saving the syscall that will be tried and its arguments in the "Next" checkpoint image, switches stack and then calls the targeted driver primitive. One of our hypotheses was that the peripheral "A" needed the peripheral "B" to be accessed. So the driver "A" primitive itself calls a driver "B" primitive, without using a kernel wrapper as we are already executing on the OS stack. At the end of driver "B" primitive the `syt_signal()` primitive is called signaling to the kernel that driver "B" device context has been modified. Then the driver "A" function finishes its work and signals also to the system that the driver "A" device context has also been modified. Returning to the kernel wrapper, the `syt_commit()` primitive is called and from the modified driver list persist into the "Next" checkpoint image the modification done to the drivers device contexts. Then the OS switches back the execution stack to the application stack and returns to the user application.

2.8 Disscussion

The mechanisms described previously lead to three possible power loss scenarios. The kernel we developed must handle each of them and be able to restore a consistent state at the start of the next lifecycle. So the power loss detection cases always consider the possibility for the kernel to fail to checkpoint the platform state in time.

Power loss during user program execution: At any point in time during the user program execution, if a power loss is detected the kernel will complete the "Next" checkpoint containing the peripherals state description with the user program executional state. This operation should be successful most of the time as the triggering threshold is configured to ensure it. The kernel will then restore peripheral state and user program state at the next boot and the user program will resume exactly at the instruction it was interrupted. Nevertheless it is still possible for the kernel to fail to checkpoint the state of the user program, for example if some peripheral is in a power mode consuming more than usual (*e.g.* radio in reception or ADC during sampling). In that case of failure the "Last" checkpoint image will be restored and the execution done in this lifecycle will be lost to ensure consistency.

Power loss during driver primitive execution: Any driver primitive is executed by the sytare kernel on a volatile stack separated from the user program execution. In these conditions, if a power loss is detected at any point during a driver primitive execution the user program state to checkpoint will be be consistent. It will be checkpointed along the peripheral state prior to the syscall and the syscall address and arguments. This checkpointing operation, if succeeded, will alow the kernel to restore the execution at the start of the concerned syscall by the next boot with consistent arguments. A failure in checkpointing will force the kernel to restore the "Last" complete checkpoint thus discarding the progress done during the lifecycle. A way to ensure the checkpoint success in the case of a power loss occuring during driver primitive execution could have been to checkpoint systematically the program state execution when switching stack, however the induced overhead could impact heavily the system performances.

Power loss during kernel execution: The only case not covered by the previously described scenarios is the occurence of a power loss detection during system operation. If such case happens the hardware interruption triggering the checkpointing mechanism stays masked until the kernel finishes its operation and returns either to the user program execution or to a driver primitive execution. The interrupt service routine will then take place as the interruptions are unmasked

when returning to the user program or to a driver primitive execution, invoking one of the previously described scenarios. Nevertheless if the platform effectively runs out of current during system operation, the execution done during the concerned lifecycle will be lost and the kernel will resume to the "Last" valid checkpoint.

These choices in the system behaviour enforces the user program execution with failure resilience given the *transiently powered system* hypothesis that the typical lifecycle duration is short compared to the program main loop execution time 1, relying on a double buffering fallback strategy under the peripheral state volatility and access atomicity problems constraints.

3 Implementation

In this section we describe our implementation of the Sytare prototype.

3.1 Hardware platform prototype

Our prototype is implemented on the Texas Instruments MSP-EXP430FR5739 FRAM Experimenter's board[1]. As the name implies, this board includes a MSP430FR5739 microcontroller. To study complex scenarios involving off-chip peripherals, we use the daughterboard connector to add a Radio Frequency transciever chip. We use the CC2500 RF chip[2] from ChipCon.

The MSP430 is a very popular architecture in the Wireless Sensor Network litterature. But more interesting to us, the FR5739 microcontroller features 16kB of embedded ferro-electric random access memory (FRAM). Thus it is a very representative example of the kind of platform we target in this work.

Actually, all NVRAM-oriented TPS papers so far either target this exact chip or the related FR5969 chip. To the best of our knowledge, no other NVRAM-based microcontroller is commercially available to date. We picked the FR5739 because its evaluation board features a daughterboard connector.

Memory architecture The resulting platform is an interesting mix of volatile and non-volatile memory. As discussed in Section 1.3, NVRAM typically offers worse performance than RAM. This is the case on the FR5739, where the 15kB of FRAM have a 125ns access time, which translates to a maximum operating frequency of 8MHz. On the other hand, the CPU, 1kB RAM, and all peripherals can run up to 24MHz, so in our experiments we set the clock frequency to 24MHz.

[1] `http://www.ti.com/lit/ug/slau343b/slau343b.pdf`

[2] `www.ti.com/lit/ds/swrs040c/swrs040c.pdf`

Power failure detection The FR5739 microcontroller embeds a voltage comparison unit (Comparator_D module) which we use to implement power failure detection. We added a voltage divisor montage composed of two resistors of 1MΩ each connected to a Comparator_D module input pin in order to monitor the in voltage drop indicating the power loss. Such a montage is shown in figure 7. This montage consumes approximatively 1.6μA and allows us to trigger checkpoint-

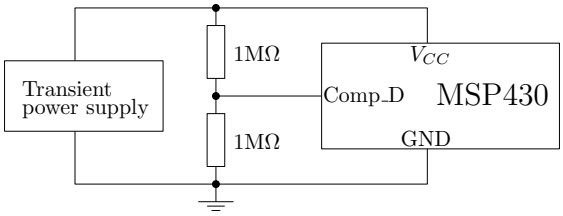


Figure 7: Montage of the plateform using the embedded Comparator_D module

ing with a theoric voltage monitoring precision of 53mV. Refering to the device datasheet the minimum execution voltage of the platform is 2.0V, so we fixed

the checkpoint trigger threshold to 2.063V. When this threshold is reached the module raises an interrupt flag and if the kernel is not running a critical operation for example accessing checkpoint content, the checkpointing routine will be engaged. If the system action cannot be interrupted the interruption will be taken into account at the end of the action.

3.2 Memory organisation

Our approach does not actually require the presence of volatile memory. An hypothetical platform with only NVRAM would be interesting to study and would lead to simpler checkpointing mechanisms. Still, because of the hardware access atomicity problem, we would have to save checkpoints of kernel state on syscall boundaries. Anyhow, our hardware platform features a volatile CPU, volatile peripherals, and also includes 1kB of RAM, so we decided to use all this volatile memory for non-persistent data.

In our prototype implementation, the Sytare kernel, the device drivers and application code are all linked into a single executable image which is transfered at once on the microcontroller. Still, we use a custom linker script to allocate various sections into distinct memory regions of the system. The resulting layout is illustrated in Figure 8 on the following page.

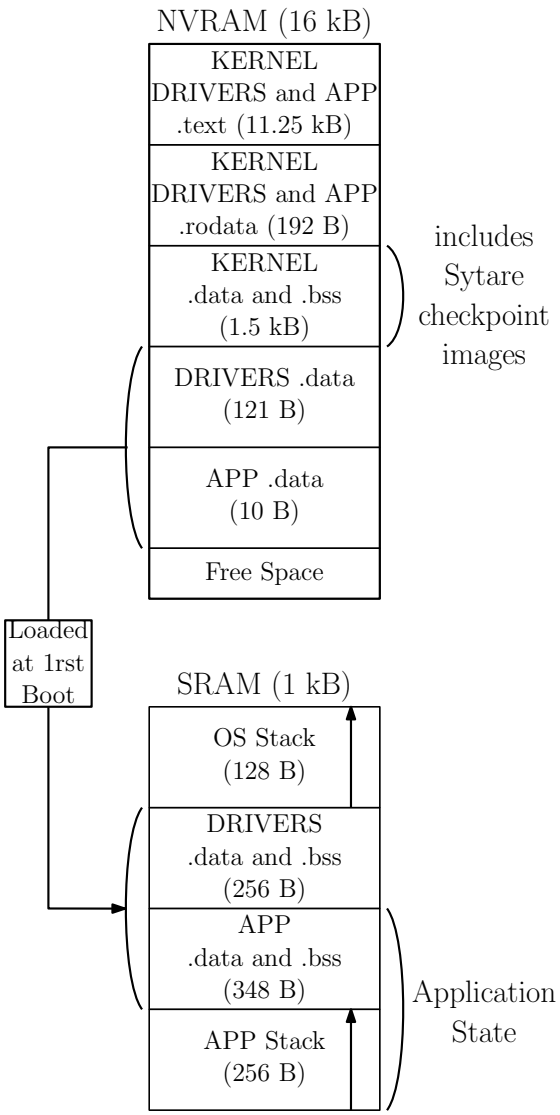


Figure 8: Sytare memory layout for WSN application: Sizes in SRAM are defined in the linker script and fixed. The size of the application and driver .text .data .rodata and .bss sections are functions of the target application, the kernel section sizes are fixed, including the checkpoint images.

3.3 System boot

When the platform is powered up, execution starts directly within the kernel. Instead of the traditional startup procedure (i.e. `crt0.s`) the code sets the stack pointer on the OS stack and jumps to the persistence management logic. Because all OS data like checkpoint images and various bookkeeping variables are kept in NVRAM, the kernel can instantaneously know what happened before the last power failure and react accordingly.

Very first boot When executing for the first time, there is no checkpoint to restore. Instead, the kernel prepares the application execution by loading its `.data` into RAM, and zeroing its `.bss` section. Then it jumps into the application's `main()` function.

Restoring a checkpoint image When booting after a power failure, we usually find a valid checkpoint image in NVRAM (otherwise we fall back to the *first boot* behaviour). The kernel loads the device contexts into RAM and then calls the `restore()` function of each driver in order to initialize the hardware peripherals to the correct state. Next the application state (i.e. `stack`, `data` and `bss` sections) are reloaded to RAM. Sytare performs all these transfers using the DMA module of the MSP430 for better performance. When the checkpoint image does not contain any syscall ID, the kernel resumes executing application code by just restoring all CPU registers accordingly: SP is set back to point to the user stack, and PC into application code.

Restarting an interrupted syscall When the checkpoint image indicates a failed syscall, then the kernel must retry it entirely. In that case, instead of returning to application code, the syscall arguments are repopulated into CPU registers and execution jumps to the driver routine. Note that the syscall entry wrapper needs not be executed again. The stack pointer does not leave the OS stack until the driver function returns, and the syscall exit wrapper switches back to the application stack.

3.4 Implementation of the syscall wrappers

When the user application needs to use the functionality of one driver, it has to invoke the corresponding syscall. For example to send a message it must call the `syt_cc2500_send_packet(msg,MSG_SIZE)` primitive instead of the original `cc2500_send_packet(msg,MSG_SIZE)` driver function. This wrapper will call the driver function with the same arguments but before that it does the following :

- save all syscall arguments on top of the user stack

- switch the stack pointer to the (volatile) OS stack
- save the program counter into the "next" checkpoint image

These operations ensure that, in case the syscall is interrupted, at next boot we will have all the right information to restart it.

While the driver function executes, it may modify the state of the underlying device. In that case, the driver calls the `signal()` function to notify the kernel. If several drivers are involved in servicing the syscall, then any number of them can signal such modifications.

When the driver function returns into the kernel wrapper, the OS does the required cleanup:

- `commit()` the modified device contexts (and only the modified ones) into the checkpoint image in NVRAM,
- switch back from the OS stack to the user stack and clean it from the stored syscall arguments,
- repopulate the registers with the return value of the driver function.

3.5 Device drivers

In this section we give details about the the device drivers implemented in the prototype system. In order of increasing complexity, the peripherals are: input/output ports, LEDs, clock system, temperature sensor, SPI controller, and RF transciever.

Simple access peripherals (type 1) *e.g.* I/O ports, LEDs. The I/O driver provides an interface to configure the external pins of the MSP430 microcontroller. Typically, each pin can be either assigned to GPIO function or connected to some peripheral module. In GPIO mode, each pin can be configured as output or as input, can be set to generated interrupts, etc etc. All these options are controlled through memory-mapped registers named *e.g.* P1DIR ("choose input or output direction for P1") or P3SEL ("select either GPIO or peripheral function for P3") etc.

On the FR5739 experimenters board, some of these I/O pins are connected to an array of 8 LEDs, so we also provide a device driver to control them. Even though both drivers have a very simple structure, the LED driver is built on top of the I/O port driver.

In the case of such simple access peripherals, adding persistence support is straightforward. As illustrated on Figure 9 on the next page, the *device context* structure of the I/O driver mimics the hardware registers exactly. The `restore()` primitive just copies its values one by one into the peripheral registers. Each driver function which actually changes something in the hardware also calls `signal()` to notify the kernel of the modification.

```
struct hw_port_registers
{
  unsigned char         out;
  unsigned char         in;
  unsigned char         dir;
  unsigned char         sel0;
  unsigned char         sel1;
  unsigned char         ie;
  unsigned char         ies;
  unsigned char         ifg;
  unsigned char         ren;
};

// sytare port driver data descriptor for persistence
struct port_device_context
{
  struct hw_port_registers    p1;
  struct hw_port_registers    p2;
  struct hw_port_registers    p3;
  struct hw_port_registers    p4;
  struct hw_port_registers    pj;
};

struct prt_device_context_t prt_device_context;
```

Figure 9: Illustration of the `device_context` type implementation for the I/O port driver.

In a traditional bare-metal program, these simple peripherals would typically be controlled directly from application code. Adding explicit devices driver and porting programs to use them does require some effort from the programmer. Also, as will be discussed in Section 4.5.2, the overhead incurred by Sytare adds a significant performance penalty to each operation.

Constrained access peripherals (type 2) *e.g.* clock system, temperature sensor, SPI controller. Each of these devices requires the software program to conform to certain rules when accessing its registers. For example, the clock system has a basic protection against accidental misconfiguration. In its normal state, all configuration registers are read-only. Before changing any of its parameters, we must "unlock" this protection by writing a certain "password" value into the first register. Then the configuration registers can be written to, and then the program has to "lock" the protection again.

Our device driver for the clock system does not directly provide access to the hardware registers but presents a higher-level interface to the user. Thus, instead

of storing the values of every registers in the *device context*, we only have a few fields describing the desired frequency and operating mode for each of the three system clocks. The other two peripherals have similar access constraints. The SPI controller requires one particular bit of a certain configuration register to be held low while changing configuration. The temperature sensor is implemented by the analog to digital converter (ADC) module, which imposes some timing constraints when initializing the hardware or when measuring a value.

In a traditional bare-metal program, these peripherals would typically be managed by dedicated device drivers and accessed only through some API. Thus adding persistence support to the corresponding code is quite straightforward. However performance-wise as for type 1 peripherals, the overhead incurred by Sytare adds a penalty to each operation.

Indirect access peripherals (type 3) *e.g.* RF transciever. The most complex device driver we implemented in Sytare controls the cc2500 radio transciever via SPI. The radio itself is quite a complex peripheral: it features a lot of configuration registers, requires certain timings on requests, and has a non-trivial internal finite state machine (the radio can be either idle, sleeping, receiving, transmitting) The driver exposes high-level primitives to the user *e.g.* send a message or put the radio to low-power sleep. It implements each of these actions via a series of SPI transactions, made through the SPI driver. Note that the `send` primitive, although it performs complex operations, let the peripheral in the same state after the call than before, and therefore does not trigger a `syt_commit()` afterwards.

4 Evaluation

The Sytare contribution is implemented by a software system running on compatible hardware in order to ensure its capabilities and evaluate the cost of the persistence service provided by the kernel. This part is build on two distinct evaluation approaches, in one hand we ran some example applications measuring the time consumption overhead induced by the sytare layer. In other hand we analyse the specific overheads induced in system operations by the sytare kernel. We also details how the time is spend during kernel operation for power loss abstraction.

4.1 Power supply

Power supply is implemented with a function generator for reproducibility. We use a square signal generator directly connected to the *Vcc* and *Ground* pins of the target board and configure the signal with various duty cycle and frequency parameters.

4.2 Metrics and variables

To evaluate the system performance we define here some performance metrics.

Baseline: continuous power For a given application program, we define T_{wired} as the time it takes to run the application from start to finish under continuous power. The "starting point" is defined as the instant power is turned on, so the duration we measure includes hardware boot time as well as program initialisation. In this experiment, we build the program with Sytare completely disabled and no persistence support. The "finish point" is defined as the instant the program reaches some arbitrary position in the code, *e.g.* encrypting a given data buffer, or send that many messages. We tuned our benchmark applications to ensure that their T_{wired} is in the right order of magnitude for a TPS, i.e. a few hundred milliseconds. We use this T_{wired} measure as a ground truth baseline for evaluating the cost of the Sytare mechanisms when running the same program under intermittent power.

Experiment: transient power When running under intermittent power, we could vary the two parameters describing each lifetime, namely the "on time" and "off time". However the "off time" is of little interest, as the platform is completely inactive in those periods. Instead we focus on the "on time", which we define as the time where the supply voltage *Vcc* is above the minimum operating threshold of the MSP430.

For each experiment, we set a certain T_{on} value and configure the power supply to repeatedly turn on for this duration and then turn off again. In each of these lifecycles, the platform boots, then the Sytare kernel restores kernel state and then the application runs until power runs out.

We define $T_{transient}$ as the time it needs for the system to reach the same executional state as in the ground truth "finish point" above. When measuring this duration we exclude all "off time" periods as they do not contribute any information to the experiement. However, we do include the boot time (hardware and software) and the cost of the checkpointing operations.

To assess the performance Sytare, we are interested in the time overhead incurred onto the execution. For a certain value of T_{on}, we define the *effective yield* Y as the following ratio:

$$Y(T_{on}) = \frac{T_{wired}}{T_{transient}} \tag{1}$$

For very small values of T_{on}, the platform will never have a chance to boot successfully and so it will never finish executing the application. In other words $T_{transient}$ would be "infinite" and the effective yield will be zero.

On the other hand, when the T_{on} duration approaches T_{wired} then the application will be able run to completion in just one lifecycle with little kernel interaction.

But even if the kernel boots only once and never has to save or restore checkpoints, execution overhead arising from the syscall wrappers still impacts performance and the effective yield will never reach 100%.

4.3 Benchmark applications

We use 4 benchmark applications with various levels of interaction with peripheral devices:

RSA This purely computational application encrypts a 128 bits data buffer with the RSA algorithm. Because it uses no peripherals but has a significant memory footprint, it allows us to study the performance of our application checkpointing mechanism.

Diode counter The program slowly counts from 0 to `max_integer` and displays the value of the counter on the platform's diodes. This simple application allows us to study the impact of the syscall wrappers on performance as well as evaluate the performance of adding persistence to simple access peripherals.

Sense and aggregate Demonstrating the use of timing constrained peripherals, this application senses the temperature 10 times using the processor ADC, stores the numbers in an array as well as the computed mean. Between each measurement, a delay of 5 milliseconds is observed.

Sense and send (WSN) Typical wireless sensor network application. It senses the temperature in the environment using the processor ADC, aggregates 10 measures and sends this information along with others (basic computations, statistics on the platform) to a wired powered sink via RF signal. The application then puts RF transciever in sleep mode and waits one second by software delay. This application demonstrates the transparent use of timing constrained peripherals.

These applications were built without the Sytare integration to measure ground truth reference measures on continuous power. These measurements are made with a CPU running at 24 MHz (configured in the first executed code lines, before program RAM initialisation).

4.4 Benchmark application evaluation

In this section we evaluate our prototype implementation on several benchmark applications. To evaluate the performance overhead induced by Sytare, we execute each application in two different settings. First, we build the program with no

persistence support at all, and execute it with continuous power. Then, we build the program with Sytare enabled, and execute it with intermittent power. We measure for each benchmark application:

- T_{on}^{min} as the minimal time to execute successfully the application in a transient power context.
- Y^{max} as the maximal yield obtained for a benchmark application corresponding theoretically to the condition $T_{on} = T_{wired}$.

4.4.1 Computational RSA application

The experiemental results for the *RSA* application are illustrated on Figure 10 and summarized below:

- $T_{on}^{min} = 2.79$ ms
- $Y^{max} = 0.98$

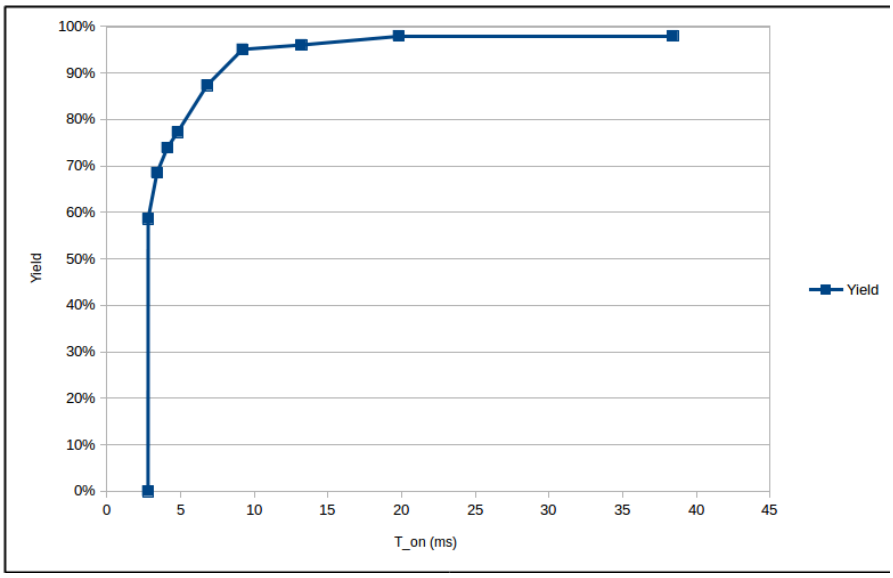


Figure 10: Temporal yield measurements in function of runtime window – RSA demo application

4.4.2 Leds counter application

The experiemental results for the *LEDs counter* application are illustrated on Figure 11 and summarized below:

- $T_{on}^{min} = 2.79$ ms
- $Y^{max} = 0.99$

This application uses only leds and so only uses one driver that will require to be persisted across power losses. The persistance of that one driver though implies no behaviour modification in the system capability to run across multiple short lifecycles.

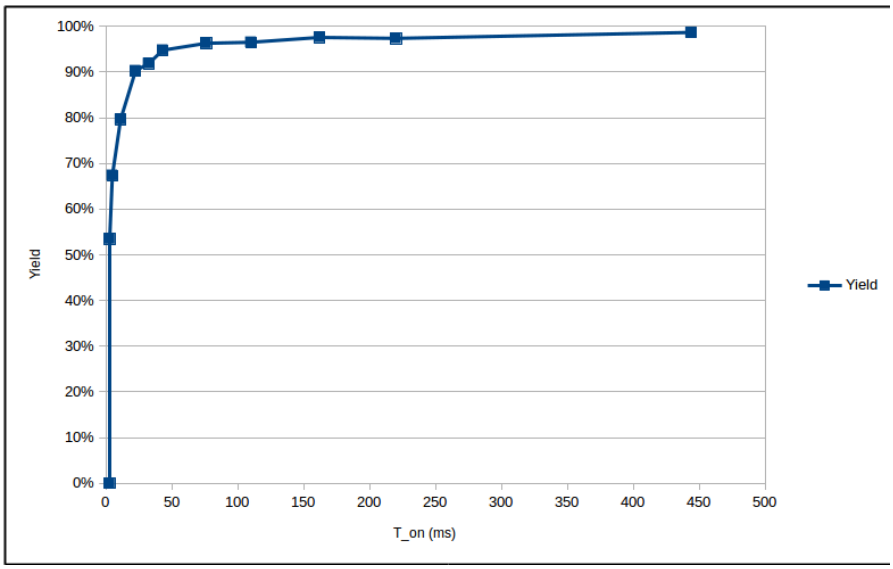


Figure 11: Temporal yield measurements in function of run time window – led counter application

4.4.3 Sense and aggregate

The experiemental results for the *Sense and aggregate* application are illustrated on Figure 12 and summarized below:

- $T_{on}^{min} = 2.90$ ms
- $Y^{max} = 0.97$

The temperature sensing application uses multiple peripherals but stays efficient in terms of minimal computation window as the different drivers used don't have long hardware initialisation. the ADC require to the application to wait for the end of its measurements, but this wait is limited and don't occur at every lifecycle.

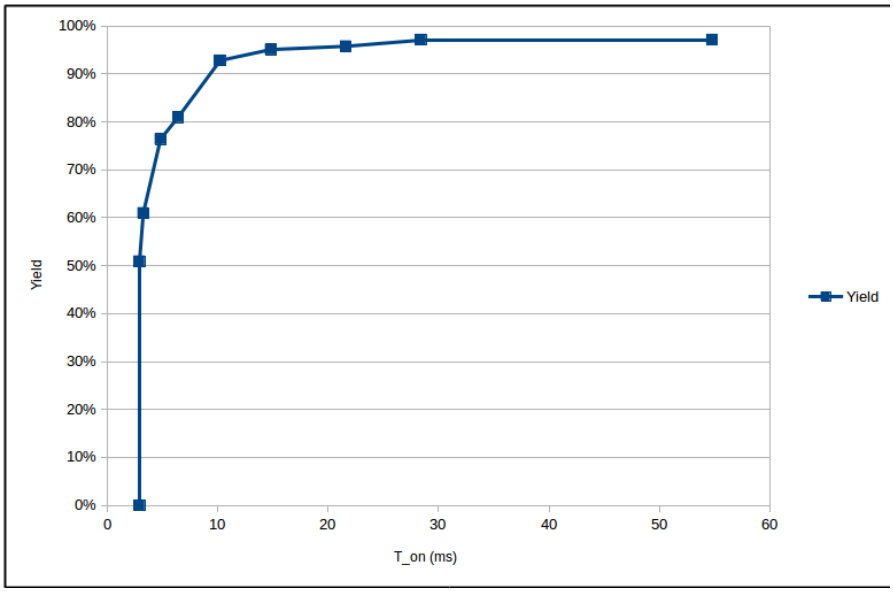


Figure 12: Temporal yield measurements in function of run time window – Sense and aggregate application

4.4.4 Sense and send (WSN) application

The experiemental results for the *Sense and Sense (WSN)* application are illustrated on Figure 13 and summarized below:

- $T_{on}^{min} = 9.40$ ms
- $Y^{max} = 0.99$

The WSN application is a realistic application used for evaluating Sytare. The addition of a complex active RF transciever increases the minimal computation window supported by the system by 3 as the RF chip initialization (or restoration) requires active polling and multiple SPI communication. Besides, the action of sending a message consumes much more current, inducing a voltage drop increasing the chance of failed checkpoint or to have to retry a syscall. Despite these remarks, our system is still able to run the application on transient power under the TPS hypothesis, validating its behaviour.

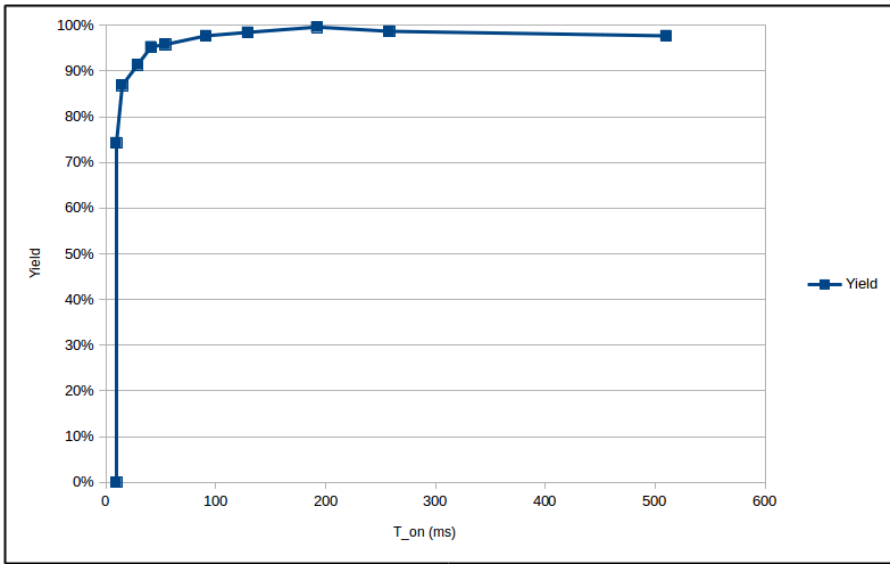


Figure 13: Temporal yield measurements in function of run time window – Sense and send application

4.4.5 Discussion

These applications all succeed in running on transient power due to Sytare integration. We observe a similar behaviour in the temporal yield computed for each one in function of the computational window given to the platform. The minimal computational window is however different between some applications depending on which peripheral they use, that the kernel will have to persist. Typically, an application using the radio frequency chip in our system is limited by the restoration function of the RF driver time consumption fixing the lower bound of the accessible computational window to approximatively 10 ms. The other benchmark applications do not use this peripheral and so their minimal computation window is around 3 ms. This demonstrates that Sytare efficiency is dependent on the addressed peripherals and their complexity.

4.5 Kernel evaluation

This section aims at describing the system timings and behaviour. First we evaluate how time is spent at system boot, and then we study the overhead of the syscall layer on the performance of driver primitives.

4.5.1 System boot

The first diagram in Figure 14 illustrates the various execution phases which compose a lifecycle. The second diagram represend how much time the system spends during boot, as described in section 3. We made these measures on the *Sense and send (WSN)* application as we wanted to show the respective restoration time of the different drivers used.

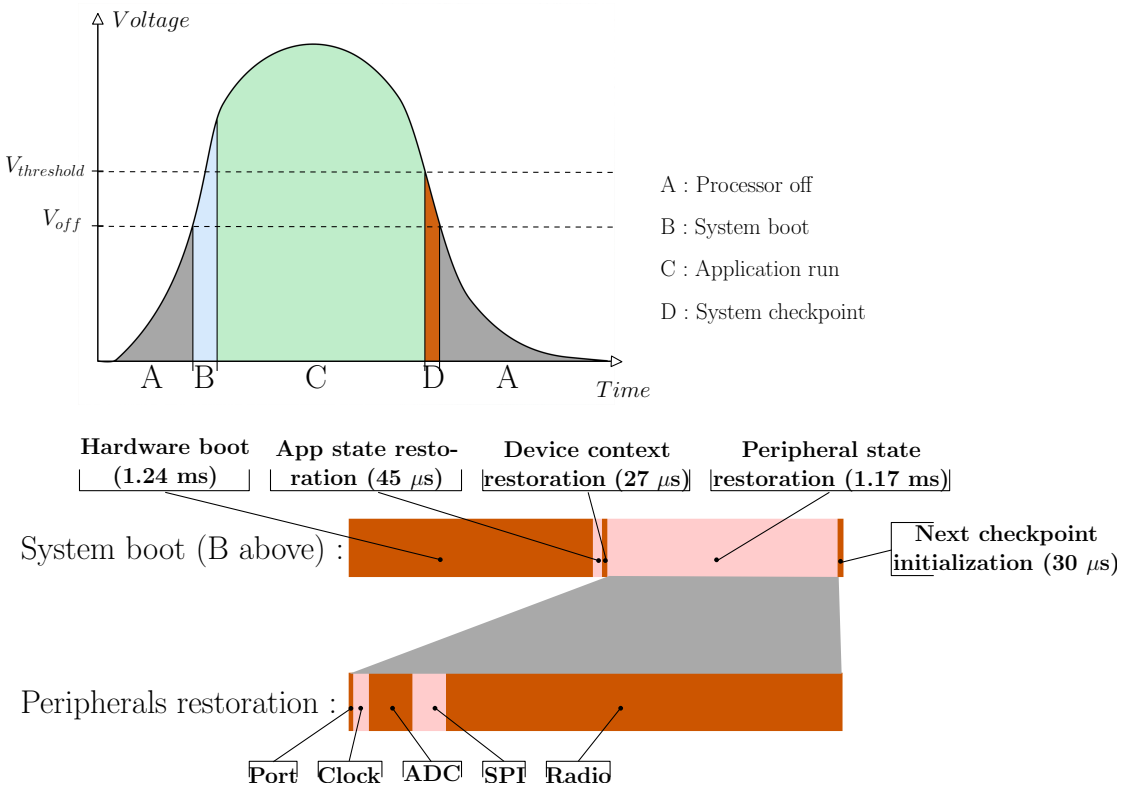


Figure 14: System boot sequence time consumption

Firstly, the hardware need to startup at power on taking a fixed amount of time. We measured 1.2 ms from the time when the power gets above the processor lower executional voltage to the first instructions the system can execute. The kernel then spends a fixed amount of time restoring the application RAM state via DMA copy. The driver structures restoration via DMA and the next checkpoint initialisation are dependent on the different drivers used as an unused driver will not be persisted. The different peripherals take a restoration time depending on the hardware to restore, on the way it can be accessed, if there is an active polling phase in the hardware startup and on the size of the configuration to restore. We can see that a significant part of the system boot time is dedicated to the hardware

state restoration by the drivers. The hardware we used was not designed to support transient execution, thus explaining the time consumption which is mostly used into hardware specific actions done by the drivers during their restoration primitive call by the kernel.

The program checkpoint phase during the power loss is done in fixed time as we dump the whole user RAM content into NVRAM. The time after checkpointing and before power loss can be seen as negligible as having a fixed checkpointing time we have tuned the $V_{threshold}$ value to have just the time to checkpoint in normal conditions. If the power loss occurs during a syscall or during system initialisation it will not be taken into account before the end of the concerned system action, possibly inducing a checkpoint fail. The tradeoff between the time granted for checkpointing and the possible checkpoint failure is not disscussed here as the involved system time (around 45 μs per lifecycle) is low in front of the drivers and peripheral restoration time (more than 1 ms).

4.5.2 Syscalls evaluation

It is important to note that the overtime induced by the kernel is not only the sum of the kernel boot and the kernel checkpoint times as during the application run, the call to drivers primitive via syscall induces time consumption by the kernel to persist the device contexts modifications. The time overhead for several drivers primitives is shown in figure 15.

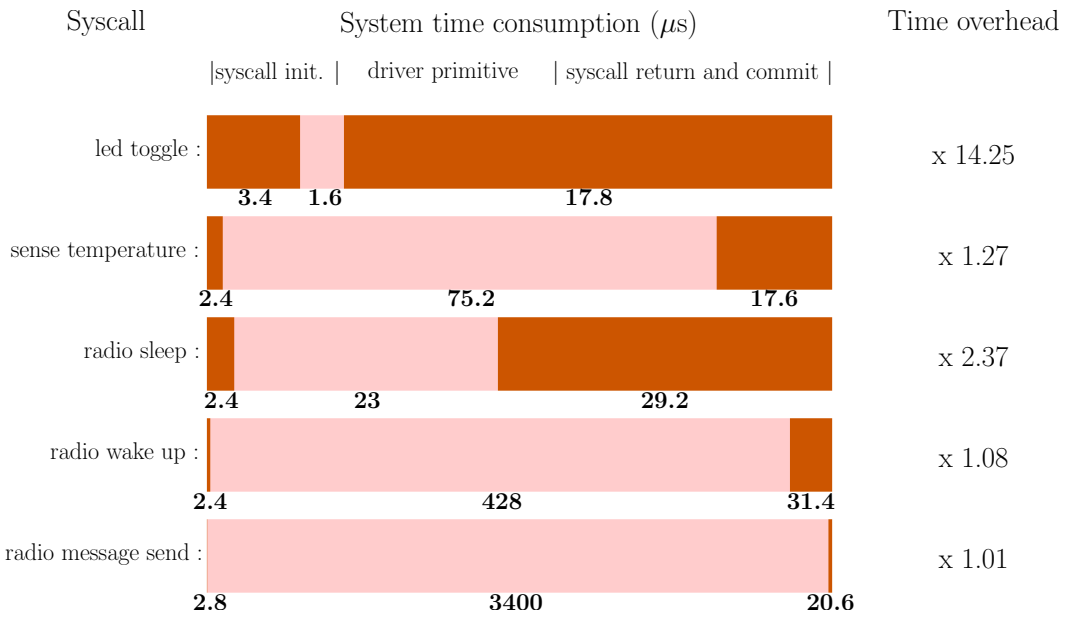


Figure 15: Kernel temporal impact on drivers primitives calls

We observe on this diagram that depending on the complexity of the accessed peripheral and the complexity of hardware actions achieved during a syscall, the overtime induced by the kernel for context switch and drivers device contexts commit varies from more than 90% of syscall time to a extremely low value (under 1%). Nevertheless the syscalls impact on the system is low as from the applicative point of view the time spent in syscall doesn't drastically impact the overtime induced by the system, mostly during restoration phase. In addition the use of long syscalls, for example the radio send message primitive, is much more visible in the system overall time as the total syscall duration is 2 orders of magnitude higher than the time used for the system to toggle the state of a pin.

4.5.3 Memory occupation

The RAM overhead of Sytare is mostly imputable to the need of a separate kernel stack. The application RAM occupation stays the same with Sytare integration as no code modification of the application is required. The drivers memory occupation into RAM is increased approximately by the size of the mirrored configuration. For example the Radio Frequency chip driver used into Sytare benchmark application used 44 additional bytes in RAM after its integration for persistence. The OS variables and checkpoints are located in NVRAM and so do not impact the RAM occupation of the system.

5 Conclusion and Perspectives

This report presents the first version of the Sytare software layer. Sytare offers services for handling software running on transiently powered systems. At time of writing, Sytare is the first tool that associates checkpointing, a classical solution to handle transient power, with a mechanism that ensures a safe use of non trivial peripherals such as timer, serial interface, ADC or radio transceiver. Sytare has been implemented on the MSP-EXP430FR5739 board from Texas Instrument which includes 15kB of FRAM together with traditional RAM. This implementation shows the impact of Sytare in terms of performance and also validates the Sytare concept on a real transiently powered system. This report studies the tradeoff between the duration of the powered periods and the additional cost of the Sytare software layer. It quantifies the impact on driver calls, it shows for instance that the time overhead induced by Sytare in radio driver calls is less than 1%.

There are many remaining issues before getting to a tool that can be used in industry. The main limitation of Sytare currently is that it does not allow user interruptions. This limitation can be overcome easily but has not been implemented yet, this is currently going on. Another important possible improvement concerns

the value of the thresholds used for checkpointing and for resuming execution. These values have an important impact on performances and are very dependent on architecture, application and possibly other factors such as temperature etc. It will probably be necessary to set up an *adaptive* threshold choice. Finally, of course, Sytare has to be tested on other platforms and other applications. The possible integration in a lightweight operating system such as RIOT or Contiki has to be studied too.

References

[AAMS14] Fayçal Ait Aoudia, Kevin Marquet, and Guillaume Salagnac. "Incremental checkpointing of program state to NVRAM for transiently-powered systems". In: *ReCoSoC 2014: 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*. 2014.

[BCGL11] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. "Operating system implications of fast, cheap, non-volatile memory". In: *HotOS 2011: 13th USENIX conference on Hot topics in Operating Systems*. 2011.

[Bel08] Gordon Bell. "Bell's Law for the Birth and Death of Computer Classes". In: *Communications of the ACM* 51 (2008), pp. 86–94.

[BKC+13] Steven Bartling, Sudhanshu Khanna, Michael Clinton, Scott R. Summerfelt, John A. Rodriguez, and Hugh P. McAdams. "An 8MHz 75uA/MHz zero-leakage non-volatile logic-based Cortex-M0 MCU SoC exhibiting 100-percent digital state retention at VDD=0V with <400ns wakeup and sleep transitions". In: *ISSCC 2013 : IEEE International Solid-State Circuits Conference*. 2013, pp. 432–433.

[BM16] Naveed Bhatti and Luca Mottola. "Efficient State Retention for Transiently-powered Embedded Sensing". In: *EWSN'16: 13th ACM International Conference on Embedded Wireless Systems and Networks*. 2016.

[BPS+08] Michael Buettner, Richa Prasad, Alanson Sample, Daniel Yeager, Ben Greenstein, Joshua R Smith, and David Wetherall. "RFID sensor networks with the Intel WISP". In: *Sensys 2008: 6th ACM Conference on Embedded Network Sensor Systems*. ACM. 2008, pp. 393–394.

[BWM+15] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. "Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems". In: *IEEE Embedded Systems Letters* 7.1 (2015).

[DGV04] A Dunkels, B Gronvall, and T Voigt. "Contiki: a lightweight and flexible operating system for tiny networked sensors". In: *29th Annual IEEE International Conference on Local Computer Networks*. IEEE. 2004.

[GPPT16] Fei Guan, Long Peng, Luc Perneel, and Martin Timmerman. "Open source FreeRTOS as a case study in real-time operating system evolution". In: *Journal of Systems and Software* 118 (2016).

[JLL+14] 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.

[JRR14] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. "QuickRecall: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers". In: *VLSID 2014: 27th International IEEE Conference on VLSI Design and 13th International Conference on Embedded Systems*. 2014, pp. 330–335.

[LBL+13] Y. Lee, S. Bang, I. Lee, Y. Kim, G. Kim, M. H. Ghaed, P. Pannuto, P. Dutta, D. Sylvester, and D. Blaauw. "A Modular 1 mm^3 Die-Stacked Sensing Platform With Low Power I2C Inter-Die Communication and Multi-Modal Energy Harvesting". In: *IEEE Journal of Solid-State Circuits* 48.1 (2013).

[LLL+15] Yongpan Liu, Zewei Li, Hehe Li, et al. "Ambient energy harvesting nonvolatile processors: from circuit to system". In: *DAC 2015: 52nd Annual Design Automation Conference*. 2015, 150:1–150:6.

[LR15] Brandon Lucia and Benjamin 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.

[MSCT14] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng. "Overview of emerging nonvolatile memory technologies". In: *Nanoscale Research Letters* 9.1 (2014).

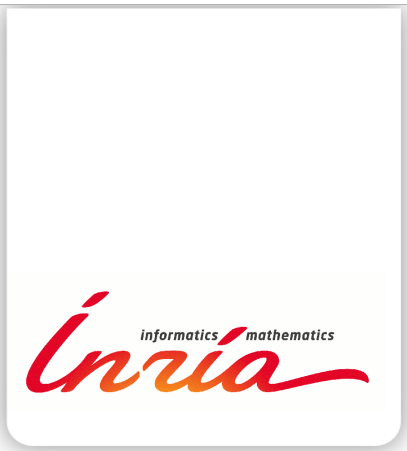
[MZL+15] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. "Architecture exploration for ambient energy harvesting nonvolatile processors". In: *HPCA'15: High Performance Computer Architecture*. IEEE. 2015, pp. 526–537.

[ODV+09] Fredrik Österlind, Adam Dunkels, Thiemo Voigt, Nicolas Tsiftes, Joakim Eriksson, and Niclas Finne. "Sensornet checkpointing: Enabling repeatability in testbeds and realism in simulations". In: *EWSN 2009: 6th European Conference on Wireless Sensor Networks*. Springer, 2009.

[RL14] Benjamin Ransford and Brandon Lucia. "Nonvolatile Memory is a Broken Time Machine". In: *MSPC 2014: ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. 2014.

[RSF11] Benjamin Ransford, Jacob Sorber, and Kevin Fu. "Mementos: system support for long-running computation on RFID-scale devices". In: *ASPLOS 2011: 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2011.

[SG10] David Stonier-Gibson. *Understanding embedded microcontroller multitasking RTOS alternatives*. date accessed: sept 2016. SPLat Controls. 2010. URL: `http://www.splatco.com/rtos_1.htm`.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399