Sytare: a Lightweight Kernel for NVRAM-Based Transiently-Powered Systems

Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, Guillaume Salagnac,
Univ Lyon, INSA Lyon, Inria, CITI, F-69621 Villeurbanne, France
firstname.lastname@insa-lyon.fr

Abstract—In a near future, energy harvesting is expected to replace batteries in ultra-low-power embedded systems. Research prototypes of such systems have recently been proposed. As the power harvested in the environment is very low, such systems need to cope with frequent power outages. They are referred to as *transiently-powered systems* (TPS). In order to execute non-trivial applications, TPS need to retain information between power losses. To achieve this goal, emerging *non-volatile memory* (NVM) technologies are a key enabler: they provide a lightweight solution to retain, between power outages, the state of an application and of its peripheral devices. These include sensors, serial interface or radio devices for instance. Existing works have described various *checkpointing* mechanisms to adapt embedded applications to TPS but the use of peripherals was not yet handled. in these works. This paper proposes a solution for embedded applications using any peripheral device to run despite transient power. We follow a kernel-oriented approach resulting in minimal impact on the programming model of the application. We implement the new concepts in our lightweight kernel called *Sytare*, running on an MSP430FR5739 micro-controller and we analyze the cost of the proposed solution.

Index Terms—Embedded Systems, NVRAM, Energy Harvesting, Low-power, Wireless Sensor Networks, Internet of Things

1 Introduction

Embedded connected tiny objects are increasingly used in our daily lives. This raises important challenges in terms of energy management and environmental issues. Under some conditions, the use of wires or batteries to power embedded systems is very inconvenient because of form factor, cost or maintenance considerations. To solve this issue, a new class of wireless, batteryless, computing devices recently appeared: *transiently-powered systems* (TPS). TPS are powered by energy harvesting, using sunlight, radio waves, temperature gradient or mechanical motion. The low level of power provided by these techniques causes very frequent power outages, typically every dozen of milliseconds, and prevents the execution of applications longer than this duration.

The use of non-volatile RAM (NVRAM) has recently been proposed as a basis to keep the program state across power outages. Several works propose program *checkpointing* techniques to save the state of the *computational part* of the system (CPU, execution stack, global variables), but do not handle the cases where the application uses peripherals (*e.g.*, LED, ADC, or RF transceiver). However, the use of such peripherals is mandatory in embedded systems and must be enabled in transiently-power systems to make these low energy systems actually usable.

In this paper, we address the problem of executing applications on top of a transiently-powered system without losing their progression between power outages. Compared to existing works, our approach allows applications to use non-trivial peripherals such as a radio transceiver. The mechanisms we propose focus on minimizing the impact on the programming model.

We identify three problems that must be addressed in order to persist peripheral state across reboots of a TPS: *peripheral state volatility*, *peripheral access atomicity* and *interrupt handling*. This paper proposes an original solution to these problems based on a *kernel* approach. All problems are addressed through the definition of a new driver API. Peripheral access atomicity is ensured by a specific mechanism for granting atomic access to peripherals. Peripheral state volatility and interrupt handling leverage new checkpointing mechanisms which persist peripheral state.

We also present an implementation of these principles: the Sytare lightweight kernel, and validate our proposal on a platform equipped with RAM and NVRAM. Compared to the former published implementation [1], the work presented in this paper includes interrupt handling which is definitely mandatory for any embedded application. The present paper also explains the solution to the problems mentioned above in a more detailed fashion. The performance of Sytare is evaluated, on this platform, against various embedded application benchmarks.

The paper is organized as follows. Section 2 provides background and related works and exposes the problems that need to be addressed when dealing with persistence of peripherals. Our proposals are detailed in Section 3, which deals with runtime with and without interrupts. Then Section 4 details our implementation of the proposed mechanisms in Sytare, and Section 5 evaluates the overhead induced by this Sytare kernel.

2 Background and Related Work

There are many fields concerned by this research. We present former works related to embedded systems, transiently-

This work is supported by Inria (Inria Project lab ZEP https://project.inria.fr/iplzep/) as well as Insa-Lyon/Spie IoT Chair.

powered systems and non-volatile memories. Then we precisely state the problems we address.

2.1 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 [2]. 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 greatly 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 or from external sources; *e.g.*, solar power, piezoelectricity, thermal gradients, or electromagnetic fields as in RFID systems.

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 [3] is an attempt to bridge the gap between RFID systems and traditional sensor networks. Similar to a sensor network node, the WISP has a few sensors connected to a programmable micro-controller. And 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 tackled the problem of miniaturizing the whole platform even more. For instance, the Michigan Micro Mote [4] is a 1.0mm^3, general purpose, ultra-low-power, configurable sensor node platform able to harvest energy from different sources and perform wireless communication. It embeds a Cortex-M0, a few kilobytes of SRAM and few kilobytes of persistent SRAM. The recent Capybara platform [5] further extends the optimization of harvesting and energy management depending on the target application.

Even when the energy source is active, the harvested power level is typically low [6] compared to what the system consumes in active mode. Storing energy in a capacitor is thus necessary to allow for useful work to be done in short bursts. For instance, contact-less 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 unfeasible. Besides feasibility issues, the constraints imposed to the programmers of these devices are very tough and releasing new software is highly-demanding in terms of engineering because of energy-related concerns.

2.2 Programming tiny embedded systems

TPS architectures form the low end of the "embedded systems" spectrum. On these systems, applications are programmed either in a bare-metal fashion, *i.e.*, without any operating system support, or on top of a very small operating system.

In the first case, the developer is directly in charge of managing the hardware peripherals as well as organizing all concurrent activities on the platform. The corresponding program structure is typically a "super loop" [7] architecture: all background tasks are performed within a single infinite loop, which itself is preempted by interrupt service routines. In the second case, application code executes on a small operating system like FreeRTOS [8] or Contiki [9], which offers limited functionality: interrupt processing, thread management, and sometimes a networking stack. In this paper, we focus on the bare-metal super loop architecture. We further make the assumption that program code can be decomposed into two distinct parts: *driver code* in charge of handling hardware accesses, and *application code* that implements the high-level logic of the system. Each time the application needs to access the hardware, it invokes a driver routine. This approach is a usual way to design embedded software. Both driver code and application code can be written either by the same developer, or by separate developers.

2.3 Non-volatile architectures

Several non-volatile memory (NVM) families are emerging [10]. This will eventually remove the distinction between slow/non-volatile "storage" and fast/volatile "memory" [11]. Current NVM technologies still suffer from slow write times, high write energy and limited write endurance. It is nevertheless promising to use these technologies, even though their place within the memory hierarchy is not definitive today. Indeed, naively replacing RAM with NVRAM has undesirable side-effects. Because power losses are frequent, they can occur in the middle of the modification of a non-volatile data structure. When the platform reboots, the program restarts with inconsistent data [12]. This issue, sometimes referred to as the "broken time machine" problem [13], is a major motivation for the work presented here.

In addition, storing data in NVRAM makes each access slower and/or more energy-expensive in comparison to volatile RAM, while storing data in RAM gives good execution performance, but brings back the problem of volatility. For this reason, most of the non-volatile architectures currently foreseen use a combination [14] of both RAM and NVRAM. For example, the MSP-EXP430FR5739 board from Texas Instruments includes 16 kB of FRAM together with volatile RAM.

Because of the broken time machine problem and the energy-related concerns, a memory architecture composed of both volatile and non-volatile cells is likely to be the most adequate and realistic architecture for transiently-powered systems. The target of this work is such a hybrid memory architecture.

Another research trend is to change the micro-architecture and make part of the processor itself non-volatile. Because of the limitations mentioned above, most non-volatile architectures still follow a hybrid approach: each CMOS register is not replaced but complemented with a *non-volatile flip-flop* [6], [15], [16], [17], [18]. These non-volatile processors are able to automatically *save* and *restore* their contents to and from non-volatile storage. But such a persistence mechanism requires a large amount of simultaneous NVRAM operations, causing significant spikes in current consumption. As a result, this approach is not applicable to the rest of the platform; *e.g.*, main memory or peripherals.

For performance reasons, program data structures are typically allocated to volatile RAM, requiring some checkpointing mechanism in order to survive power losses. This issue is discussed in Section 2.4. In any case, both non-volatile architectures and program checkpointing techniques address the same issue; *i.e.*, the volatility of program state. In this paper, we focus on another, distinct issue presented in Section 2.5: system peripherals such as sensors or I/O devices require additional, more sophisticated techniques. We consider the two approaches complementary. Our contributions could equally be implemented on top of a non-volatile processor.

2.4 Program checkpointing

Checkpointing consists in saving the program state to a non-volatile memory device and restoring it if an issue occurs. Checkpointing has been widely used for crash recovery of critical systems for instance. We are interested in checkpointing within the context of low power embedded devices powered by harvesting. Figure 1 shows an ideal scenario [19] where checkpoints are performed just before power loss.

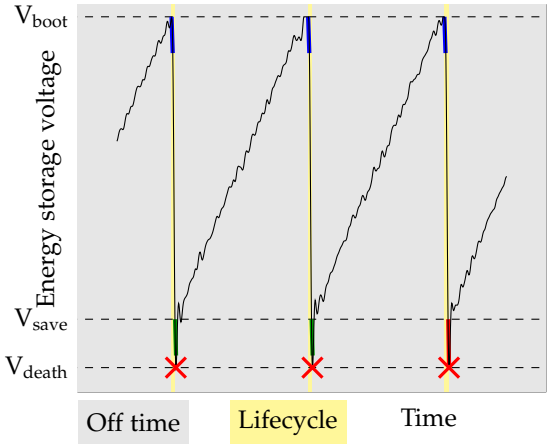

Fig. 1: Typical On/Off cycles of platforms supplied by energy harvesting. Checkpoint operations can succeed (green) or fail (red). Application and devices are always restored in a coherent state.

The first paper about checkpointing for transiently-powered systems introduces Mementos [20], a software runtime for the the Intel WISP [3]. To prevent progression loss, Mementos periodically interrupts the application and measures the remaining energy level. If the energy level is below a certain threshold, Mementos saves the CPU registers and copies the contents of RAM to flash memory. Mementos makes power loss transparent to the application program, but it is designed specifically for flash memory of the WISP and does not easily generalize to NVRAM.

Hibernus [21] proposes to save program state to FRAM, which yields significant time and energy savings compared to Mementos. Instead of interrupting the application on a periodic basis, Hibernus uses a hardware device to trigger the checkpoint operation only 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 power losses is a promising strategy. Jayakumar *et al.* [22] also propose checkpointing based on hybrid memory and power loss detection. Another work, from Bartling *et al.* [14], proposes a design for a non-volatile micro-controller with 10 kB of ROM, 8 kB of SRAM, and 64 kB of FRAM. Upon detecting a power loss, the chip automatically saves all CPU and peripheral registers to FRAM, approximately 320 bytes of data.

Checkpointing can be optimized to reduce its impact on performance. Aït-Aoudia *et al.* [19] propose an incremental checkpointing scheme to reduce the amount of NVRAM writes as much as possible. Bhatti and Mottola [23] 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 sub-optimal.

None of the papers mentioned above address the issue of peripheral state persistence. Phoenix [24] proposes a checkpointing mechanism which allows to roll back the system state in case of a driver failure. The problem addressed is different: their goal is to survive driver faults and not power outages. The volatile memory of the system will not be inconsistent. Our approach is different from Phoenix but we agree that "the resource-constrained nature of embedded systems presents [...] unique challenges to adapting existing checkpointing mechanisms".

In a recent paper [25], Wang *et al.* propose to implement part of the checkpointing mechanism directly inside the hardware. In order to speed up the restoration of the RF transceiver, they design and implement a dedicated SPI controller with non-volatile logic. This approach is complementary to our work: while it enables to restore the transceiver state in parallel with the rest of the system, it does not guarantee peripheral data consistency, and specific care is required to ensure atomicity. In the following Section, we discuss this issues in more detail.

2.5 Problem statement

The problem we address in this paper is to make, for transiently-powered systems with NVRAM, hardware peripherals *persistent* across reboots so that the application does not have to handle power loss.

In all embedded systems, peripherals are key components. Their complexity ranges from simple access peripherals such as LEDs, buttons and GPIO, to more complex peripherals that impose a specific access sequence to their registers (timers, ADC, serial ports for instance) or even indirect access peripherals that must be accessed through another peripheral, such as a radio chip accessed through SPI serial link. Many peripherals interact with the application through interrupts.

Section 3 decomposes the problem into three aspects: *peripheral state volatility*; *peripheral access atomicity*; and *interrupt handling*. We first propose a solution for handling peripheral devices while setting aside interrupts. Then we focus on how to handle interrupts. We briefly present each of these three aspects, respectively in Sections 2.5.1, Section 2.5.2,and Section 2.5.3.

2.5.1 Peripheral state volatility problem

The first issue is how to cope with the volatility of peripheral state. Because peripherals configuration can be complex, capturing and restoring the internal state of peripherals might

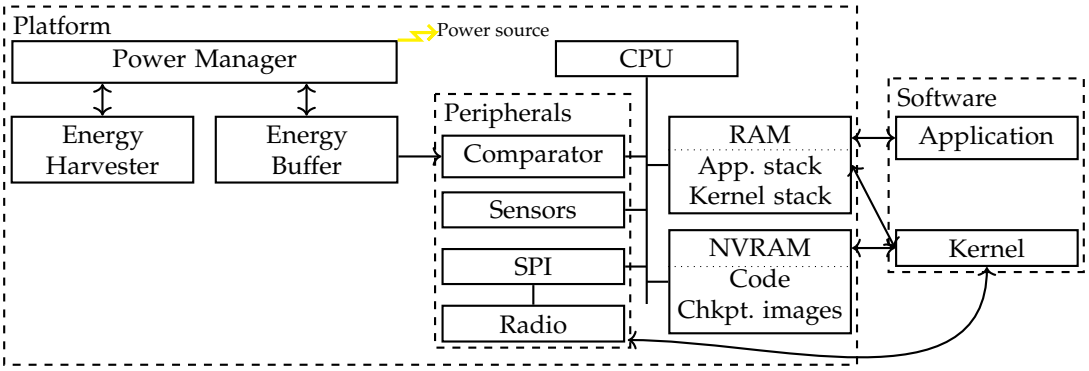

Fig. 2: Overview of the hardware and software architectures targeted by this proposal.

require more complex techniques than simply restoring their state from NVRAM to RAM.

For instance, if the radio chip is configured for listening to incoming packets, the new configuration should be restored after a power outage has occurred. As mentioned before, this issue is not solved in existing works on transiently-powered systems. Our approach leverages a strong assumption: peripherals are considered entirely volatile. Although non-volatile flip-flops are heavily studied and even used in some specific research fields such as neural networks [26], no peripheral circuit (clock, SPI bus, *etc.*) embeds non-volatile memory today. This field, as mentioned before, is subject to recent investigation [25]. Note that solutions exist, in the literature, to handle state volatility in the specific case of mass storage devices using transactions [12].

2.5.2 Peripheral access atomicity problem

The second issue is how to cope with power loss occurring in the middle of a hardware request being serviced. For instance, consider a scenario where the application sends a radio packet. If a power loss happens in the middle of the transmission, it would not make sense, at the next boot, to send the rest of the packet.

2.5.3 Interrupt handling problem

An interrupt signals that an event has occurred and that the software (kernel, driver or application layer) must handle it. Three problems arise when dealing with interrupts in the context of transiently-powered systems.

First, an interrupt handler may modify data in memory without the application being aware of it. This may lead to inconsistency issues within application state. Second, handling an interrupt may change not only data in memory but also some peripheral states, leading to inconsistency issues between the actual peripheral and what the application assumes it is. These two problems are not new [27], [28], [29], but we need to address them in the specific context of NVRAM-based transiently-powered systems. Third, interrupts carry some data such as, for instance, the origin of the interrupt or the data located in peripheral memory. This piece of data is entirely volatile and will no longer be available upon hardware reboot.

In Section 3, we address the peripheral state volatility and atomicity problems, as well as interrupt handling.

3 Handling peripherals in a TPS

Figure 2 shows an overview of both hardware and software architectures leveraged by this lightweight kernel proposal. The micro-controller can access both RAM and NVRAM since both memories are mapped.

Let us assume that we start from a classical checkpointing mechanism of kernel for transiently-powered systems such as the one explained in Mementos [20], *i.e.*, without peripheral handling and interrupt support.

The state machine of the system is sketched in Figure 3. When a power loss happens, the kernel saves a copy of the *application state* from RAM to NVRAM. When power comes back on, it checks whether a valid checkpoint exists and restores the application state in RAM.

However, this scenario is valid only if power loss occurs while executing application code. If power loss occurs during the execution of driver code, then at the next boot it will not make sense to *resume* its execution exactly where it stopped. This highlights the *access atomicity* problem mentioned above. We propose to modify the structure of the code so as to handle peripheral state persistence.

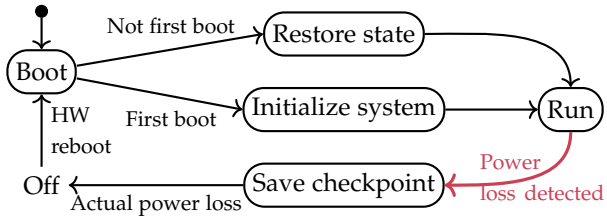

Fig. 3: State machine of a simple transiently-powered system with a classical checkpointing kernel.

3.1 Modifying bare-metal embedded software

In the state machine of Figure 3, there is no distinction between application code and driver code. This is usually the case for *bare-metal* code; *i.e.*, an embedded application without operating system. As mentioned before, we make the assumption that this code can be split into two types of code: *application code*; and low-level *driver code*. If this is not the case, *i.e.*, if memory-mapped hardware register accesses are spread along application functions, then code refactoring

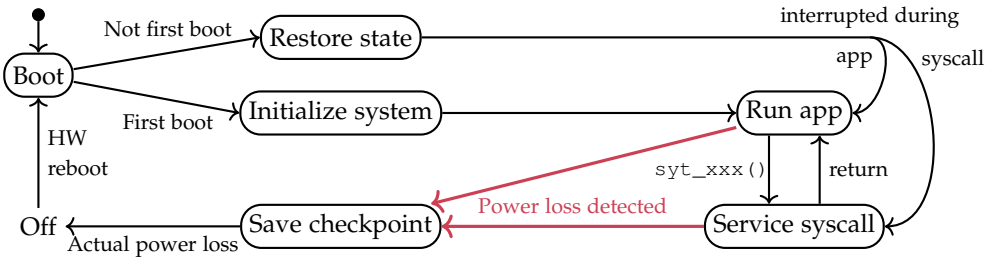

Fig. 4: State machine of a Sytare TPS including the checkpointing logic as well as the system call wrappers described in Section 3.3. For simplicity, interrupts are not illustrated here.

will be needed to clearly separate application code and driver code.

We expect *application code* to encompass higher-level functions as well as library code. As such, we define the *application state* to be composed of all the global variables of these modules, the contents of the execution stack (local variables, control flow) and CPU registers.

Driver code consists of all functions which provide access to hardware features. 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.

We propose to add a third level of code which we call *kernel code* that will be inserted at the frontiers between application code and driver code and hence, encapsulate each driver function call. *Kernel* code is responsible for persistence management, which includes saving and restoring application state to and from non-volatile memory. The kernel code adds a thin wrapper around each driver function. The wrapper contains the necessary features to address the atomicity problem. We call these wrappers *system calls*. In our proposed solution, the application should always invoke a driver function through a *system call*, which is part of the kernel. It means that the application should never directly call a driver function, as it would bypass the wrapper, but it should call the corresponding *system call* instead. This might induce some simple code refactoring by the developer.

In addition, the kernel also includes procedures for checkpointing (saving state to NVRAM), restoring (loading state from NVRAM), and initializing the platform. From now on, for the sake of clarity, we will refer to all the software components of the kernel code (system calls, initialization, checkpointing and restoring code) as *Sytare* which is the name given to our particular implementation of these solutions. Of course there are many other ways to implement our proposal, but the main principles stated here need to be respected.

Sytare kernel code follows a state machine which slightly differs from that of Figure 3 as shown in Figure 4. The restoration procedure now acts differently depending on whether the power loss occurred during a system call or during application code. The kernel *wrappers* allow Sytare to make this distinction. To what extent this helps in providing persistence and atomicity for peripherals is explained further in Sections 3.2 and 3.3.

Like other similar checkpointing mechanisms [20], [19], a hardware device detects imminent power loss and interrupts the application. Section 4.1 details our implementation of this mechanism. The checkpoint procedure then copies application state to a non-volatile data structure, called *checkpoint image*.

Sytare maintains two checkpoint images, in a double-buffer fashion. The last valid image is kept intact while the next image is built incrementally. This way, if a power loss occurs during a checkpoint operation, the system will be able to recover the last valid image at the next boot.

3.2 Solution to peripheral state volatility

As discussed in Section 2.5, checkpointing memory contents is not enough when the system includes hardware peripherals. Restoring the state of a hardware device 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.*. The solution to this issue is quite simple but requires some cooperation from the driver developer. This cooperation consists in storing the state of the device in a *device context* and in implementing `restore()` and `save()` functions for each driver.

3.2.1 Device context

For simple peripherals, the state of the hardware device is reflected in driver code, such as the value of control registers for instance. If the device has a more complex state space; *e.g.*, a finite state machine with several control modes, then the driver usually keeps track of the current mode within the device context.

In Sytare, we require this kind of data to be explicitly encapsulated in a so-called *device context* data structure. This change is illustrated in Figure 5 on a simple LED device example. Thanks to these *device context* structures, the kernel can take a snapshot of the state of each device at various points in time and augment the checkpoint image with peripheral state, hence providing peripheral state persistence.

All device drivers offer some `init()` primitive which performs the correct initialization procedure. Sytare requires each driver to provide an additional `restore()` primitive which will be called upon restoring a checkpoint.

At boot time, the kernel restores all device contexts to memory, and then successively invokes the `restore()` function of each driver. 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 simply calling `init()`. However for more complex peripherals

(a)
```
void led0_switch_on() {
  HW_REGISTER_FOR_THIS_LED=1;
}
```

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

led_context_t *led_context;

void led0_switch_on() {
 HW_REGISTER_FOR_THIS_LED=1;
 led_context->led_state[0]=1;
}
```

Fig. 5: Illustration of the device context data structure. (a) Straightforward initial driver routine for a LED. (b) Modified driver routine, compliant with Sytare kernel requirements. Note that only the device context type is defined in the driver code, the device context *allocation* is managed by the kernel.

which require indirect access and/or impose certain access constraints, the `restore()` function uses the information restored from the saved device contexts.

If some driver A uses services of some other driver B, the kernel must ensure that the `restore()` function of driver A is called only after B has been restored. The dependencies between devices are usually static and the kernel should be able restore the devices in an order compatible with these dependencies. In our implementation, we simply ask to the application developer to manually order the calls to the `init()` routines of the drivers.

3.3 Solution to peripheral access atomicity

To solve the peripheral access atomicity problem, we have to make peripheral accesses atomic. If a driver operation is interrupted, then it must be considered entirely failed as the kernel cannot ensure state consistency at lower granularity. Our proposed solution uses the *system call* wrapper mechanism defined above to treat differently application code and driver code.

3.3.1 Volatile and non-volatile stacks

The contract between the application and the kernel is that a *system call* will be executed entirely before being registered in the checkpoint image. If a power loss happens in the middle of a system call, then at the next boot the system call will be *retried completely* instead of just *resumed*. Our implementation uses different memory regions for application execution environment (*i.e.*, the application stack) and driver execution environment (*i.e.*, the kernel stack).

Application code and system calls have different checkpointing policies when it comes to the variables: system call variables must not be saved since we want them to be retried from the beginning in case of power loss. To do so, Sytare separates execution stack of application and system calls. This so-called *kernel stack* (or *volatile stack*) is never included in the checkpoint image, which guarantees that any partial progress inside a driver is volatile. Stacks are switched in the system call wrapper invoked by the application. The wrapper also saves a pointer to the system call and a copy

```
void syt_led0_switch_on(void)
{
  // backup arguments and switch to kernel stack
  syt_syscall_ctx_switch();
  // call original driver function
  __asm__ __volatile__("call #led0_switch_on");
  // save() device contexts and switch to app. stack
  syt_syscall_return();
}
```

Fig. 6: Illustration of the system call wrapper for the simple driver routine `led0_switch_on` from Figure 5.

of its arguments into the checkpoint image. This piece of information is sufficient to retry a system call if a power loss occurs during the driver function execution. Once the data are saved, the wrapper then calls the actual driver function which runs on the *volatile* stack.

An example of such system call, *i.e.*, wrapper of a driver function, is shown on Figure 6 as it is implemented in Sytare. The macros `syt_syscall_ctx_switch()` and `syt_syscall_return()` implement stack switching. They depend on the underlying hardware and contain assembly code, but one can notice that writing the wrapper for each driver function is quite systematic and may be done automatically.

The `syt_syscall_return` is also responsible for saving the corresponding device context into the next checkpoint image. By ensuring that `syt_syscall_return` is the only function that saves the device contexts, we ensure to always have a coherent device state during restoration. Note that there is a specific treatment for nested system calls; *i.e.*, driver routines that call other driver routines. Saving device contexts is performed only when the outer driver routine returns. This is illustrated further in Figure 10.

3.3.2 Power loss scenarios

Having this solution in mind, there are two possible scenarios. If the system call is interrupted by a power loss, the checkpoint operation has to be triggered, but it only has to save application state, since the *former* state of the peripheral has already been saved to non-volatile memory during *former* system call. At the next boot, the kernel restores application state to memory and can re-run the system call that has been interrupted with its correct arguments.

In the other case, where the system call successfully returns, the kernel wrapper erases the saved system call pointer and arguments from the checkpoint image, calls the `save()` function of the relevant drivers (see Figure 7) and switches back to application stack.

Note that partial progress in the driver routine may have side effects, for instance when a radio packet has been sent but power loss occurred before the system call returned. In such cases, we make the assumption that the situation is not handled by device drivers but by higher software layers. Network and communication protocol layers should not be part of the kernel but should be considered application code. In the specific case of sending multiple radio packets, this must be handled by the application code and by providing an adequate radio device context. Very recent work has started addressing this problem [30].

```
void led_save(void) {
    if(led_device_dirty) {
        memcpy(
            &next_image->led_device_context,
            led_device_context,
            sizeof(*led_device_context));
    }
}
```

Fig. 7: Illustrating the `save()` primitive for the led driver. The `if` statement prevents the device context from being copied if the system call did not alter the device context.

Finally, to support the case where no system call happens during a lifecycle, or if all system calls are read-only operations, the kernel initializes at boot time the checkpoint image with a copy of all device contexts from the last image.

3.3.3 Checkpoint image structure

To sum up our proposal for peripheral state persistence, we show the content of a complete checkpoint image in Figure 8. The checkpoint image contains information describing the application state as did former works related to checkpointing, but it also contains the driver state and the kernel state, ensuring atomicity of system calls.

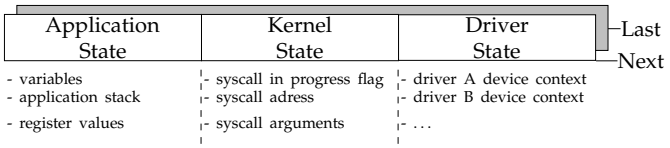

Fig. 8: Illustration of the checkpoint image content.

3.4 Solution to interrupt handling

Until now, we have only considered sequential execution of code. Interrupt handlers behave as concurrent pieces of code and bring additional consistency issues when power loss occurs.

3.4.1 Top-half and bottom-half

A standard approach for handling interrupts in traditional operating systems consists in splitting interrupt handlers into two parts, the *top-half* and the *bottom-half* [31], [32]. When an interrupt occurs, it is immediately handled by the operating system that executes a so-called *top-half* routine with interrupts disabled. This routine usually acknowledges the interrupt and registers a deferrable *bottom-half* in charge of handling the lengthy operations associated with the interrupt.

We propose a natural extension of this approach to handle transient power. Our proposal leverages the *top-half* being part of the kernel code and the *bottom-half* being application code, as in a classical operating system.

As mentioned in Section 2.5.3, we have to pay attention to several issues, which are well-known in the embedded system area. A top-half may modify peripheral state and a bottom-half may modify peripheral or application state. This situation leads to consistency issues between the actual peripheral state and what the application assumes it is.

Furthermore, interrupts carry data, such as the origin of the interrupt or the data located in the peripheral generating the interrupt. We cannot afford to lose the volatile data on power loss, because the application would use incoherent data after the platform has rebooted. This problem is specific to our TPS context. In the following of this Section, we describe our solutions to these problems.

3.4.2 Interrupt handling design concerns

In order to keep the interrupt-based power loss detection mechanism, interrupts must be enabled as long as possible. Hence, while interrupts are disabled during top-halves, they must be kept enabled during bottom-halves. This leads to a choice: bottom-halves can or cannot be nested; *i.e.*, could a bottom-half run in the middle of another bottom-half? To simplify the complexity of the resulting kernel, we chose to forbid nested bottom-halves and we propose instead to sequence them with the creation of a *queue of bottom-halves* scheduled in a FIFO fashion.

Another important assumption is that no system call will occur in top-halves. This makes sense because top-halves belong to kernel code. Hence, when a top-half needs to access to a peripheral, it directly calls the original driver function.

The application developer is likely to request peripheral access from a bottom-half. And of course, bottom-halves are only allowed to access peripherals through system calls, in exactly the same way as application code would.

3.4.3 Persistence of interrupt data

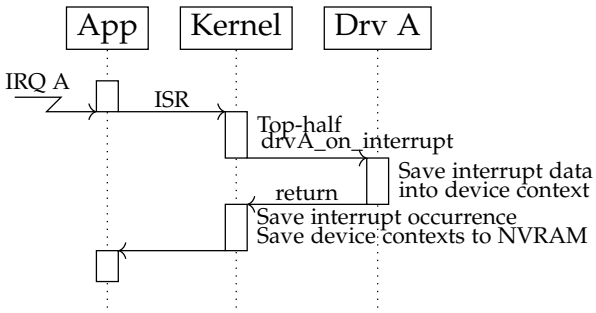

Fig. 9: Sequence diagram showing how top-halves perform interrupt data persistence.

We identify two kinds of interrupt data that must be persisted: scheduling data and peripheral data. Scheduling data contain the queue of bottom-halves waiting to be run. Peripheral data are the data carried along with the interrupt. It can be the content of a device register or a more complex structure, as a long radio packet for instance.

In order for the kernel to keep these pieces of information across potential hardware reboots, the checkpoint image depicted in Figure 8 must be extended to support interrupt data persistence.

We extend device contexts to store interrupt peripheral data. Again, there are many design choices for this extension: how many pending interrupts do we allow for a particular device for instance? In our implementation, the choice is made to limit to one the number of *simultaneous* pending interrupts from the same device. Other choices can be made and will imply more complex device context handling.

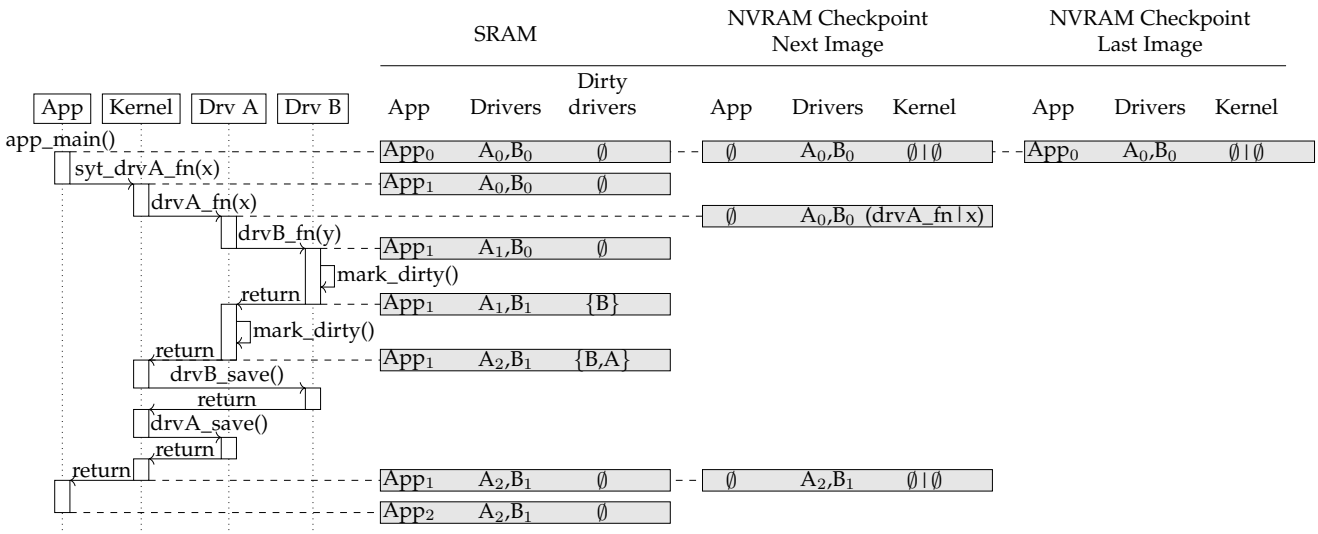

Fig. 10: Sequence diagram of a complex, *i.e.*, nested, system call with SRAM and NVRAM kernel data structures content.

To store interrupt data, Sytare requires that the driver developer provides an `on_interrupt()` routine, for each driver, that will be called by top-halves. This routine must perform the relevant copies from peripheral memory into its device context. Since power loss could happen at any moment, we want the volatile interrupt data to be extracted and copied into the device context as soon as possible, as shown in Figure 9. Once the bottom-half counterpart is scheduled, the kernel saves all modified contexts and interrupt occurrence. Then, the bottom-half will be eventually executed, possibly after a power loss.

3.4.4 Data race conditions

The last problem concerns concurrency between bottom-halves and application. Bottom-halves may access the same data as the application would access and this may result in various race condition scenarios. As a result, memory may become inconsistent for the application.

Again, the solution to this problem is very dependent on implementation choices. Race conditions between application code and bottom-halves, as well as race conditions involving accesses to peripherals from interrupt context, must be handled.

This problem is usually addressed with interrupt priorities and critical sections. For instance, it may be left to the application developer to disable interrupts (including or excluding power loss interrupt) during peripheral accesses whenever they think peripheral consistency would be threatened by interrupts. This is the solution provided in Sytare.

3.5 An example of complex system call

We illustrate our checkpointing mechanism in Figure 10 where we detail the successive operations that ensure coherent reboots whenever a power outage occurs.

The "SRAM" column describes the volatile state which includes the application state, the device contexts and a list of modified drivers with respect to the last checkpoint image built by the kernel. The "NVRAM Checkpoint Next Image" and "NVRAM Checkpoint Last Image" columns describe the checkpoint image double-buffer.

Initially, the application state is App_0 and peripherals A and B respectively have states A_0 and B_0. The application requests access to peripheral A, which has to be done through the kernel wrapper (prefixed by `syt_` as implemented in Sytare). Between the beginning of the `app_main()` function and calling `syt_drvA_fn()`, the state of the application in SRAM changes from App_0 to App_1, as every instruction impacts the volatile state of the system. At this point, peripherals A and B and their associated drivers are still in states A_0 and B_0.

The application calls the wrapper `syt_drvA_fn(x)`. Before the wrapper calls the actual driver function `drvA_fn(x)`, the kernel records that the application developer wants to execute `drvA_fn()` with parameter `x`, in order to be able to restart the call if power loss occurs in the meantime. When `drvA_fn()` calls `drvB_fn()`, the state of peripheral A has changed to state A_1 in SRAM.

Then `drvB_fn()` runs to completion, making peripheral B change to state B_1. Just before ending, `drvB_fn()` function indicates that driver B is *dirty*, *i.e.*, its state has changed compared to the NVRAM recorded device context. Then `drvB_fn()` returns back to `drvA_fn()`, which runs to completion and makes peripheral A change to state A_2 and indicates driver A as dirty.

At this point in time, both drivers of peripherals A and B are dirty. Driver routine `drvA_fn()` eventually returns to the kernel wrapper, which calls the `save()` routine of all drivers. In this specific scenario, only drivers A and B will copy their device contexts into the checkpoint image as they are the only dirty drivers. When the kernel wrapper returns to the application, the drivers are no longer dirty. In addition, the address of the system call and its parameter are cleared. Eventually, the application keeps on making progress, changing its state to App_2.

One can check that, whenever power loss occurs and triggers application state copy to the next checkpoint image, the next image presents a coherent state of the system: application and drivers.

4 IMPLEMENTATION

In this Section we describe our current Sytare implementation [33] and we evaluate its performance and overhead on the Texas Instruments MSP430FR5739 micro-controller which contains 16 kB of NVRAM. Sytare source code is available online[1].

4.1 Hardware

The Texas Instrument MSP430 is a very popular architecture in the Wireless Sensor Network literature. The MSP-EXP430FR5739 FRAM Experimenter's board[2] includes 16 kB of embedded ferroelectric random access memory (FRAM). Most NVRAM-based TPS papers so far either target this exact chip or the related FR5969 chip. To the best of our knowledge, no other NVRAM-based micro-controller family is commercially available yet. We also used the CC2500 Radio Frequency transceiver daughterboard [3] from ChipCon.

The MSP-EXP430FR5739 platform is an interesting mix of volatile and non-volatile memory: 16 kB of FRAM which has a 125 ns access time, and 1 kB RAM which can, as all peripherals, run up to 24 MHz. In our experiments we set the clock frequency to 24 MHz.

Power loss detection is performed using the on-chip voltage comparator and an external voltage divider, as in [21]. It raises an interrupt so that periodic polling is not required.

4.2 Memory organization and kernel boot

In the Sytare prototype implementation, device drivers and application code are all linked using `gcc` linker into a single executable image which is transferred at once onto the target platform. The RAM and NVRAM memory organization (*.text*, *.data* and *.bss* sections for kernel, driver and application code, the two stacks and room for checkpoint image) is configured using a linker script as it is usually done for embedded systems.

When the platform is powered on, kernel execution starts. The kernel stack is reset afresh. All Sytare data, such as checkpoint images and various bookkeeping variables, are kept in NVRAM. Hence, the kernel knows what happened before the last power loss and act accordingly.

4.3 Checkpointing and restoration

When an imminent power loss is detected and raises an interrupt, the kernel takes over and copies application stack, `.data` and `.bss` sections from volatile memory into the next checkpoint image. All data sections are copied, not only the modified sections like in [19]. The use of the Direct Memory Access of the platform makes the process simple and fast. This operation always copies the same amount of data. Thus, checkpointing time does not depend on factors such as the power loss frequency. In our case, checkpointing always lasts less than 26 µs.

On boot, Sytare repopulates all volatile sections based on checkpoint image information. In more details, on the very first boot, it consists in initializing `.data` and `.bss` sections. On regular boot after a power loss, it consists in restoring application stack, `.data`, `.bss` and peripherals configuration.

1. https://gitlab.inria.fr/citi-lab/sytare-public/
2. http://www.ti.com/lit/ug/slau343b/slau343b.pdf
3. www.ti.com/lit/ds/swrs040c/swrs040c.pdf

4.4 Implementation of the system call wrappers

When an application needs access to any driver routine, it invokes the corresponding system call. For example to send a message it calls the `syt_cc2500_send_packet(msg,size)` primitive instead of the original `cc2500_send_packet(msg,size)` driver function. The implementation of system call wrappers follows exactly what is depicted in Section 3.3.

If several drivers are involved in servicing a system call, then all of them will mark themselves dirty. Of course, when a driver routine returns a value, the return value is copied as the return value of the wrapper, in order to grant the application with the information exposed by the driver API.

4.5 Device drivers persistence in Sytare

We hereafter give some details about the device drivers implemented Sytare.

4.5.1 Simple access peripherals

The I/O driver provides an interface to configure the external pins of the MSP430 micro-controller. Each pin can be either assigned to GPIO function or connected to some peripheral module. It can also be configured as output or as input, and can be set to generate interrupts, *etc.*. All these options are controlled through memory-mapped registers. For instance, P1DIR is used to choose input or output direction for Port 1.

In the case of such simple access peripherals, adding persistence support is straightforward. The *device context* structure of the I/O driver mimics the hardware registers exactly as shown formerly in Figure 5. The `restore()` primitive simply copies its values into the peripheral registers.

4.5.2 Constrained access peripherals

The clock system, SPI bus and temperature sensor must comply to either a specific access protocol or timing constraints. For example, the clock system has a basic protection against accidental misconfiguration. The software must unlock the clock system by writing a password to a specific register, perform the required operations and then lock it back again.

In a traditional bare-metal program, these peripherals are already managed by dedicated device drivers and accessed only through some high-level API. Thus adding persistence support to the corresponding code is quite straightforward. The driver developer provides a high-level driver API and Sytare provides high-level system call API in order to handle the constraints of those peripherals. However, the mechanisms added by Sytare to ensure the persistence of peripheral states add a penalty to each operation.

4.5.3 Indirect access peripherals

The most complex device drivers implemented in Sytare use indirect access. For instance, the CC2500 radio transceiver is controlled via the SPI bus. The radio itself is quite a complex peripheral: it features many configuration registers, requires certain timings on requests, and has a non-trivial

internal finite state machine (the radio can be either idle, sleeping, receiving and transmitting). The driver exposes high-level primitives to the application developer; *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. Again, Sytare provides system calls corresponding to this high level API.

5 Evaluation

We now evaluate Sytare performance on the MSP430-EXPFR5739 board.

5.1 Power supply

Power supply is implemented with a programmable power supply for reproducibility. It generates a square signal that can support hundreds of mA for peak activity, even though in our applications the current needs never get higher than 25 mA. The output of our programmable power supply is directly connected to the supply pins of the target board. It can make lifecycles as small as 3 ms. In our experiments, we make the duration of the lifecycles vary, but within the same experiment, it is kept constant. However our programmable power supply is capable of running scenarios with various lifecycle durations.

5.2 Metrics and variables

To evaluate Sytare performance, we define some metrics. For a given application, we define T_{wired} as the time it takes to run entirely the application under continuous power, without Sytare or any persistence mechanism. The starting point is the instant when power is turned on, so the duration we measure includes hardware boot time as well as program initialization. The finish point is the instant the application completes. We use this T_{wired} measure as a ground truth baseline for evaluating the cost of Sytare mechanisms. In order for the concepts of starting and finish points to make sense, we built the benchmark applications in a slightly different way than bare-metal super loop. Instead of having an infinite loop, the applications iterate a fixed amount of times. We also define two parameters in relation with power supply, namely on-time and off-time. However the off-time is of little interest, as the platform is completely inactive during those periods. Hence, we focus on the on-time, which we define as the duration of the supply voltage being above the minimum operating threshold of the platform.

For each experiment, we set a certain lifecycle duration value, called T_{on}, and configure the power supply to repeatedly power on the platform for this duration, then power it off. We define $T_{transient}$ as the time needed for the system to reach the same execution state as in the ground truth finish point above. When measuring this duration we exclude all off-time periods as they do not give any information. However, we do include the boot time (hardware and software) and the cost of the checkpoint operations.

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

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

For small values of T_{on}, recall that T_{on} is kept constant for one transient execution, the platform will never have a chance to boot successfully and the application will never finish. In other words $T_{transient}$ would be "infinite" and the effective yield would be zero.

On the other hand, when T_{on} approaches T_{wired}, then the application will run to completion in just one lifecycle with little kernel interaction. In this case, the overhead corresponds to the system call wrappers as there is no checkpoint operation, nor restore operation, nor multiple hardware boot times. So even if the kernel boots only once and never has to save or restore checkpoints, execution overhead arising from the system call wrappers still impacts performance and the effective yield will never reach 100%.

5.3 Benchmark applications

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

RSA encrypts a 128-bit data buffer using 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.

LEDs slowly increases a counter and displays its value using LEDs. This simple application allows us to study the performance overhead of adding persistence to a simple access peripheral and system call wrappers.

Sense senses the temperature 80 times using the internal ADC, stores the values in an array as well as the average value. Between each measurement, the application waits 5 milliseconds. The application demonstrates the use of constrained peripherals.

WSN computes the average temperature over 10 measurements and sends the result using the radio chipset. The whole process is done 50 times. This application also demonstrates the transparent use of constrained peripherals for more realistic scenarios.

All applications can be compiled on top of Sytare or as bare-metal applications. We measure for each benchmark application:

- T_{wired}, the average execution time of the application, without Sytare and without power loss, over 100 samples. The standard deviation is given in the results.
- T_{on}^{min}, the smallest lifecycle duration for which the application can be executed to completion in a transient power context. We observe, in Figure 11, that $Y(T_{on})$ behaves like a hyperbola. Since the slope of $Y(T_{on})$ becomes straight when T_{on} is small, T_{on}^{min} cannot be measured with great accuracy. Hence the values we show are computed using the hyperbola model $\hat{Y}$, by solving the equation $\hat{Y}(T_{on}^{min}) = 0$.
- Y^{max}, the maximal yield obtained for a benchmark application; *i.e.*, when $T_{on} \geq T_{wired}$.

The results are given in Table 1. The LED counter application uses only LEDs, thus only uses one driver that must be persisted across power losses. Although the driver is simple, the overhead of Sytare is visible in the yield ($Y^{max} = 99.53\%$).

The Sense application uses multiple peripherals but stays efficient in terms of yield ($Y^{max} = 99.94\%$) as the different drivers used do not have a long hardware initialization.

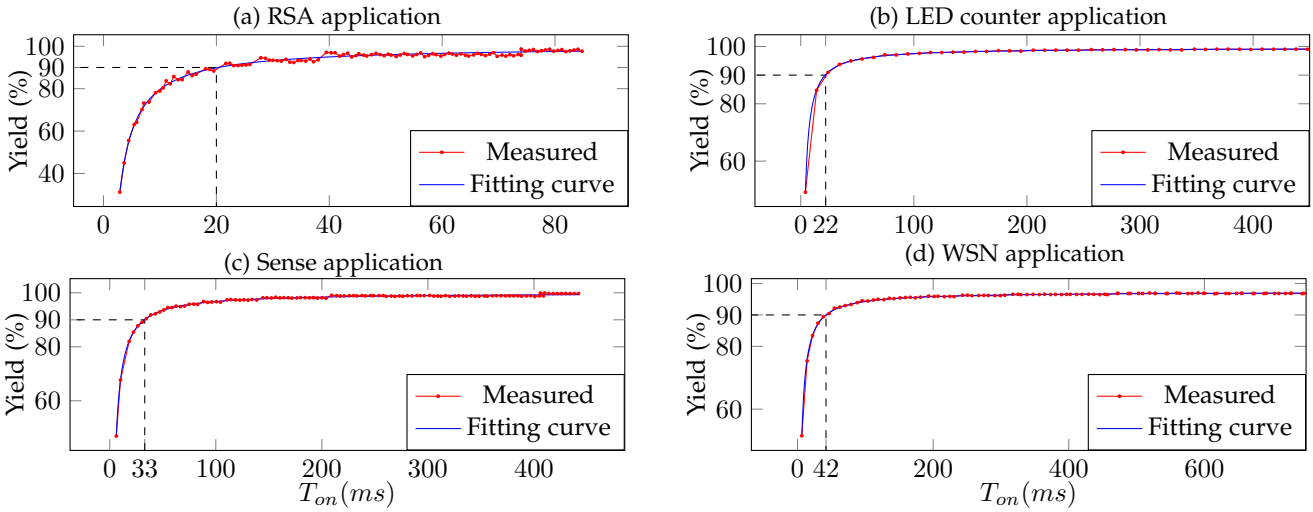

Fig. 11: Temporal yield measurements as a function of lifecycle duration (T_{on}) for all applications.

App.	T_{wired}	Std dev.	Y^{max}	T_{on}^{min}
RSA	73.61 ms	0.25 ms	0.9986	1.95 ms
Leds	1003.61 ms	5.06 µs	0.9953	2.04 ms
Sense	410.56 ms	4.61 µs	0.9994	3.27 ms
WSN	891.80 ms	0.36 ms	0.9729	2.70 ms

TABLE 1: T_{wired}, Y^{max} and T_{on}^{min} for each application. *Std dev.* is the standard deviation of T_{wired} over 100 samples.

The WSN application is a realistic application used for Sytare evaluation. The addition of a complex active RF transceiver decreases the maximal yield of the application, as RF chip initialization and restoration require active polling and multiple SPI communications. Besides, the action of sending a message consumes much more current, inducing a voltage drop and increasing the risks of failed checkpoint or of having to retry a system call. Despite these points, our system is still able to run the application on transient power under the TPS hypothesis. The system call overhead is rather low since the application reaches 97,29% yield efficiency.

In addition, Figure 11 tells us that the applications can perform 90% yield efficiency for short lifecycles. For instance, WSN application achieves 90% yield with lifecycles of 42 ms. In comparison to the 891 ms given by T_{wired}, the requirements to ensure a certain quality of service are less constraining than in the ground truth without checkpointing.

The conclusion on benchmark evaluation is that all applications succeed in running on transient power thanks to Sytare integration. Applications can run over much shorter power periods without losing much yield efficiency.

5.4 Kernel evaluation

This Section analyzes in detail the behavior of Sytare: boot sequence and system calls. In addition, we evaluate the minimal lifecycle duration, and discuss memory-related and energy-related issues.

5.4.1 System boot

Table 2 shows the time spent by the kernel in various phases of the boot sequence. Values were measured based on the WSN application. It can be seen that a great portion of boot time is dedicated to restoring the peripheral state and that software mechanisms (checkpointing and restoration) have a low impact in comparison. In addition, our experiments show that restoring peripheral state does not take the same amount of time as it depends on the amount and type of the peripherals. An active polling phase in the hardware startup, or the size of the configuration to restore, are factors that impact boot time. More precisely, the majority of the time spent in restoring the state of peripherals is taken by the restoration of the radio device (around 78%), far above the ADC converter (8%), the SPI bus (6%), the clock (4%) and the ports (3%).

	Boot hard-ware	Restore app.	Restore device context	Restore periph. state	Init chkpt.
Time	1.24 ms	45 µs	27 µs	1.17 ms	30 µs

TABLE 2: Booting sequence of *WSN* application.

Those numbers can be compared to the time needed to checkpoint the program state during the power loss: 26 µs. Note that the V_{save}, as defined in Figure 1, has been set experimentally to a value large enough to allow checkpointing completion in normal conditions. Dynamically setting V_{save} may be important for optimization of transiently-powered systems.

5.4.2 System calls evaluation

It is important to note that the overhead induced by the kernel is not only the sum of the kernel boot and the kernel checkpoint times. During application execution, the calls to driver primitives via system calls induce kernel time overhead. This overhead is shown in Table 3 for several drivers primitives.

Primitive	Time overhead
LED toggle	+1325%
Sense temperature	+27%
Radio sleep	+137%
Radio wake up	+8%
Radio message send	+1%

TABLE 3: Kernel temporal impact on driver primitives.

We can see that the overhead induced by the kernel for context switch and for saving device contexts depends on

the complexity of the peripheral to be accessed and the complexity of hardware actions achieved during a system call. It varies from more than 90% of system call time to an extremely low value (under 1%). Nevertheless, the impact of system calls on runtime is low, as yield values can reach up to 97%. In addition, the longer the driver routine, the more negligible becomes the system call wrapper. Indeed, system call wrappers add an overhead of 1% for radio send packet routine, while they add an overhead of 1325% for LED toggle routine.

5.4.3 Minimal lifecycle duration

Except for the first lifecyle, any lifecycle encompasses a restoration and a checkpointing phase. If a lifecycle lasts less than restoration time and checkpointing time, application cannot make any progress. In addition, our solution to peripheral access atomicity problem is to make system calls atomic so they are re-run from the beginning if interrupted. Thus, depending on the application, the minimal duration of a lifecycle must be long enough for the longest or most energy-consuming system call.

Table 1 also shows the minimal durations of a lifecycle, called T_{on}^{min}, for various applications. Applications that are not peripheral intensive, such as RSA and LEDs, can make progress with shorter lifecycles than applications such as Sense and WSN. This demonstrates that Sytare efficiency dependends on the peripherals used and their complexity.

5.4.4 Memory occupation

The RAM overhead of Sytare is mostly imputable to the need of a separate kernel stack of size limited to 128 bytes. Details on kernel stack are given in Section 5.4.5. The driver memory occupation in RAM is increased approximately by the size of the mirrored configuration. For example the RF chip driver we used requires 44 additional bytes in RAM for persistence management. The kernel variables and checkpoint images are located in NVRAM and do not impact kernel RAM occupation.

App.	RSA	LED	Sense	WSN
Total NVRAM use	7366	6952	8696	11994
Checkpoint image	584	385	405	503

TABLE 4: NVRAM requirements, in bytes, of some applications. Checkpoint image size is included in the total NVRAM use.

Table 4 shows the amount of NVRAM used by each application. The checkpoint image size is the size of a single checkpoint image. Since Sytare uses a double-buffer mechanism for checkpoint images, the total NVRAM space used solely by the checkpoint images is twice as large. NVRAM requirements vary from an application to another because they do not use the same amount of drivers, of variables, *etc.*.

5.4.5 Kernel stack utilization

Sytare kernel runs on its own stack, which means that part of the memory is dedicated to the kernel and is not usable from application space. In order to evaluate the amount of used kernel stack, we use two applications: a *buttons* application that counts the amount of times the button is pressed, and a *radio* application that counts the amount of received packets Both applications display their counter on the LEDs and are based on interrupts.

Power losses	IRQ occurrences	App. buttons	App. radio
0	0	42	58
2	0	42	62
0	≥ 10	62	100
2	≥ 10	68	102

TABLE 5: Kernel stack utilization, in bytes, for buttons and radio applications, under several runtime scenarios.

We ran these applications with two parameters:

- Power losses: amount of power losses throughout the experiment.
- IRQ occurrences: number of times the application was interrupted, excluding power loss detection.

Table 5 shows how much kernel stack is actually used for both applications under these scenarios. Those results show that interrupt handling requires few dozens of bytes at most. Given these results, in our current implementation, we statically set the amount of RAM assigned to kernel stack to 128 bytes.

However, the nature of the application may change the stack requirements of the kernel, since using different peripherals may result in a different kernel stack utilization. For instance, our GPIO driver does trivial operations, while our radio driver is built on top of our SPI driver, using the GPIO driver. As a result, there are many nested calls and kernel stack usage increases. In addition, interrupts add stack utilization overhead as they involve driver calls and manipulate kernel internal functions. Power losses do not substantially change kernel stack usage, as the kernel stack is reset on boot.

5.4.6 Application stack utilization overhead

Application stack utilization mainly depends on the application itself. However Sytare needs to store some data on the application stack to achieve persistence.

When switching from application stack to kernel stack to run a system call, thirteen 16-bit registers are pushed onto the application stack. As a result, the system call mechanism adds a 26-byte overhead to application stack utilization. Since a bottom-half can run when the main application is interrupted during a system call, and since bottom-halves also can use system calls, then in this worst-case scenario the 26-byte overhead can be accumulated, making this overhead reach 52 bytes.

When power loss detection interrupt is generated, the micro-controller pushes two 16-bit registers onto the current stack. Hence, whenever power loss detection occurs while running application code, a 4-byte overhead will be observed in the application stack.

As a result, the typical worst-case scenario as far as application stack utilization overhead is concerned is when the main application is interrupted during a system call and power loss detection occurs when the bottom-half uses a system call. In this specific situation, overhead adds up to 56 bytes. In our experiments, 256 bytes for the application stack were enough in total, for all applications.

5.4.7 Energy overhead

Sytare was tested against the AEM10940 energy harvester from *e-peas*[4] company. It consists of a solar energy harvester and a power manager, and stores energy in a capacitor. In our experiments, we use a 330 µF capacitor. The embedded power manager powers on and off the MSP-EXP430FR5739 board, respectively whenever the capacitor voltage goes beyond $V_{boot} = 3.21$ V or below $V_{death} = 2.93$ V, which gives an energy budget of 313 µJ. As discussed in [5], the storage should be large enough to run the largest atomic section. Our largest atomic section is the radio packet sending system call which takes 100 µJ with this energy harvester. The solar cell we use is the CBC-SEH-01, a 32 cm^2 solar cell. A smaller cell can be used and would simply decrease the frequency of lifecycles, as long as the delivered power is large enough to enable harvesting.

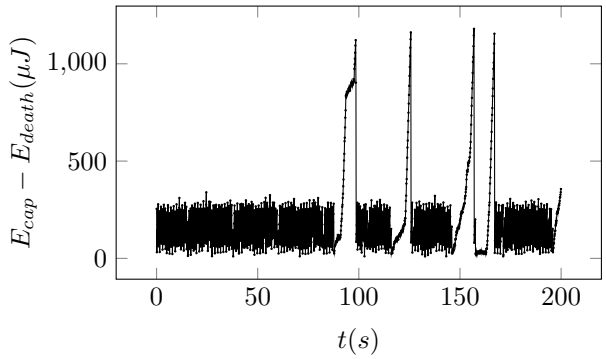

Fig. 12: Available energy in the storage capacitor, supplied by harvesting solar energy. Peaks above 1000 µJ are caused by an internal feature of the harvester when the storage capacitor is charged too slowly, when the solar cell was covered.

Figure 12 shows the amount of energy stored in the capacitor over a period of 200 seconds, under indoor light conditions, while covering the solar cell occasionally. The capacitor is refilled within 730 milliseconds in case of usual office light conditions, and within 10 seconds when the solar cell is covered. This scenario is thus realistic for an Internet of Things application solely supplied by energy harvesting.

Checkpoint creation and restoration operations respectively consume 0.183 µJ and 2.115 µJ every lifecycle, leaving 310 µJ for the application to run every lifecycle.

6 Conclusion

This paper presents the Sytare kernel. 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 for the MSP-EXP430FR5739 board from Texas Instruments which includes 16 kB of FRAM together with traditional RAM. This implementation shows that the impact of Sytare in terms of performance is reasonable and validates our proposal on a real transiently-powered system. This paper studies the trade-off between the duration of the powered periods and the additional cost of Sytare software layer. It shows that, for powered periods as small as a few milliseconds, realistic applications can make progress despite frequent power outages. In addition, the overhead of the persistence mechanisms introduced in this paper is low, in comparison to the improvement brought by substantially less constraining lifecycle durations. This paper also quantifies the impact on driver calls, and it shows for instance that the time overhead induced by Sytare in radio driver calls is less than 1%.

An important future work concerns the value of the thresholds used for checkpointing and resuming execution. These values substantially impact performance. They depend on architecture, application and possibly other factors such as temperature. It will probably be necessary to set up an *adaptive* threshold choice. It would become possible to dynamically adapt the threshold with an improved communication between the power manager and the application. In addition, the needs of an application are never constant, especially for low-power embedded applications that usually run under several power modes, such as CPU-active, low-power sleep, radio-active, *etc.*. Hence, it is interesting to analyze, either statically or dynamically, the energy requirements of applications so that the kernel can adjust the voltage threshold values.

4. https://e-peas.com/

References

[1] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Peripheral state persistence for transiently-powered systems," in *Global Internet of Things Summit (GIoTS)*. IEEE, 2017.

[2] H. Jayakumar, K. Lee, W. S. Lee, A. Raha, Y. Kim, and V. Raghunathan, "Powering the Internet of Things," in *International Symposium on Low Power Electronics and Design (ISPLED)*. ACM, 2014.

[3] M. Buettner, R. Prasad, A. Sample, D. Yeager, B. Greenstein, J. R. Smith, and D. Wetherall, "RFID sensor networks with the Intel WISP," in *Conference on Embedded Network Sensor Systems (SenSys)*. ACM, 2008.

[4] 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," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 1, 2013.

[5] A. Colin, E. Ruppel, and B. Lucia, "A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2018.

[6] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015.

[7] D. Stonier-Gibson, "Understanding embedded microcontroller multitasking RTOS alternatives," SPLat Controls, 2010, date accessed: sept 2016.

[8] F. Guan, L. Peng, L. Perneel, and M. Timmerman, "Open source FreeRTOS as a case study in real-time operating system evolution," *Journal of Systems and Software*, vol. 118, 2016.

[9] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki: a lightweight and flexible operating system for tiny networked sensors," in *International Conf. on Local Computer Networks (LCN)*. IEEE, 2004.

[10] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng, "Overview of emerging nonvolatile memory technologies," *Nanoscale Research Letters*, vol. 9, no. 1, 2014.

[11] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory," in *Conference on Hot topics in Operating Systems (HotOS)*. USENIX, 2011.

[12] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes, "Memory Management Techniques for Large-scale Persistent-main-memory Systems," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, 2017.

[13] B. Ransford and B. Lucia, "Nonvolatile Memory is a Broken Time Machine," in *Workshop on Memory Systems Performance and Correctness (MSPC)*. ACM, 2014.

[14] S. Bartling, S. Khanna, M. Clinton, S. R. Summerfelt, J. A. Rodriguez, and H. 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 *Intl. Solid-State Circuits Conf. (ISSCC)*. IEEE, 2013.

[15] F. Su, W. Chen, L. Xia, C. Lo, T. Tang, Z. Wang, K. Hsu, M. Cheng, J. Li, Y. Xie, Y. Wang, M. Chang, H. Yang, and Y. Liu, "A 462GOPs/J RRAM-based nonvolatile intelligent processor for energy harvesting IoE system featuring nonvolatile logics and processing-in-memory," in *Symposium on VLSI Technology (VSLIT)*. IEEE, 2017.

[16] Y. Liu, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M. Chang, S. John, Y. Xie, J. Shu, and H. Yang, "Ambient energy harvesting nonvolatile processors: from circuit to system," in *Design Automation Conference (DAC)*. ACM, 2015.

[17] Y. Liu, Z. Wang, A. Lee, F. Su, C. Lo, Z. Yuan, C. Lin, Q. Wei, Y. Wang, Y. King, C. Lin, P. Khalili, K. Wang, M. Chang, and H. Yang, "4.7 A 65nm ReRAM-enabled nonvolatile processor with 6x reduction in restore time and 4x higher clock frequency using adaptive data retention and self-write-termination nonvolatile logic," in *International Solid-State Circuits Conference (ISSCC)*. IEEE, 2016.

[18] Y. Wang, Y. Liu, S. Li, D. Zhang, B. Zhao, M. Chiang, Y. Yan, B. Sai, and H. Yang, "A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops," in *Proceedings of the ESSCIRC*, 2012.

[19] F. Aït-Aoudia, K. Marquet, and G. Salagnac, "Incremental checkpointing of program state to NVRAM for transiently-powered systems," in *International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE, 2014.

[20] B. Ransford, J. Sorber, and K. Fu, "Mementos: system support for long-running computation on RFID-scale devices," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2011.

[21] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, 2015.

[22] H. Jayakumar, A. Raha, and V. Raghunathan, "QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers," in *International Conference on VLSI Design (VLSID)*. IEEE, 2014.

[23] N. Bhatti and L. Mottola, "Efficient State Retention for Transiently-powered Embedded Sensing," in *International Conference on Embedded Wireless Systems and Networks (EWSN)*. ACM, 2016.

[24] R. Smith and S. Rixner, "Surviving Peripheral Failures in Embedded Systems," in *USENIX Annual Technical Conference*. USENIX, 2015.

[25] Z. Wang, F. Su, Y. Wang, Z. Li, X. Li, R. Yoshimura, T. Naiki, T. Tsuwa, T. Saito, Z. Wang, K. Taniuchi, M. Chang, H. Yang, and Y. Liu, "A 130nm FeRAM-based parallel recovery nonvolatile SoC for normally-off operations with 3.9x faster running speed and 11x higher energy efficiency using fast power-on detection and nonvolatile radio controller," in *Symposium on VLSI Circuits*, 2017.

[26] Y. Wang, L. Xia, M. Cheng, T. Tang, B. Li, and H. Yang, "RRAM Based Learning Acceleration," in *Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. ACM, 2016.

[27] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser, "Dingo: Taming Device Drivers," in *European Conference on Computer Systems (EuroSys)*. ACM, 2009.

[28] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[29] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric, "Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers," in *Intl. Conference on Automated Software Engineering (ASE)*. ACM, 2015.

[30] F. Aït-Aoudia, M. Gautier, M. Magno, O. Berder, and L. Benini, "Leveraging Energy Harvesting and Wake-Up Receivers for Long-Term Wireless Sensor Networks," *Sensors*, vol. 18, no. 5, 2018.

[31] 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," *IEEE Trans. on Computers*, vol. 60, no. 12, 2011.

[32] P. Regnier, G. Lima, and L. Barreto, "Evaluation of Interrupt Handling Timeliness in Real-time Linux Operating Systems," *SIGOPS Operating Systems Review*, vol. 42, no. 6, 2008.

[33] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Peripheral State Persistence For Transiently Powered Systems," INRIA, Research Report RR-9018, 2017.

Gautier Berthou holds a Master's degree from KTH Royal Institute of Technology as well as a Master's degree from engineering school INSA Lyon, both in computer science. After one year working as an engineer on a research project dedicated to NVRAM management in transiently-powered systems, he started working as a PhD student in the Socrate team of INRIA/CITI lab.

Tristan Delizy is a PhD student in the CITI Laboratory of INRIA / INSA-Lyon. After a master's degree in computer science (2013) from INSA-Lyon, he spent several years in industry as an embedded software developer. He then joined INRIA to work on the transiently powered systems for a year, and since 2016 he started a PhD on Dynamic Memory Management for Heterogeneous Memory Embedded Systems. His research interests include Transiently Powered Systems, Dynamic Memory Management, Normally-Off Computing and Non-Volatile Memory technologies.

Kevin Marquet is an associate professor at INSA Lyon, member of the CITI Lab and the INRIA Socrate team. He holds a PhD from University of Lille (2007) where he designed a Java OS for embedded system. He previously worked as post-doctoral researcher at Verimag, Grenoble, where he studied the formal verification of SystemC programs. His research works still revolve around operating systems and memory management.

Tanguy Risset is a full professor at INSA Lyon and a member of the CITI Lab, he is head of the INRIA Socrate team. He worked previously at Irisa Rennes on high level synthesis, then in ENS-Lyon where he created the INRIA COMPSYS team. He studied software defined radio, created the SOCRATE INRIA Team at INSA Lyon and was co-leader of the FIT/CorteXlab platform. His research interests now include Ultra-low power systems, Non-volatile memory, IoT and software defined radio.

Guillaume Salagnac is an associate professor at INSA Lyon and a member of the CITI lab since 2009. He holds a PhD (2008) and an MSc (2004) both from University of Grenoble (France), where he studied automatic dynamic memory management for real-time embedded Java applications. Before joining the CITI lab, he was a post-doctoral researcher at the CSIRO ICT Centre in Brisbane, Australia, where he worked on high-level programming for Wireless Sensor Networks. His current research interests lie in the area of programming languages and operating systems for resource-constrained embedded platforms.