

RICE UNIVERSITY

Moving Device Power Management Out of Drivers

by

Jie Liao

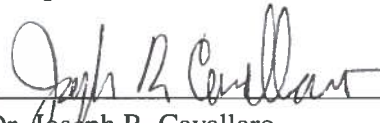
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

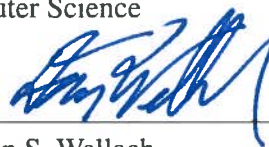
APPROVED, THESIS COMMITTEE:



Dr. Lin Zhong, Chair
Professor,
Electrical and Computer Engineering and
Computer Science



Dr. Joseph R. Cavallaro
Professor,
Electrical and Computer Engineering and
Computer Science



Dr. Dan S. Wallach
Professor,
Computer Science and
Electrical and Computer Engineering

Houston, Texas

November, 2016

ABSTRACT

Moving Device Power Management Out of Drivers

by

Jie Liao

Device drivers are well-known to be complex and error-prone. A widely practiced strategy toward simpler drivers is to move functions out of drivers. Power management (PM), an important function provided by drivers, however, has long resisted this strategy. Today, device drivers provide functions to manage device PM context. This work presents our attempt to move PM context management out of drivers and realize it in a device-independent manner. Our key idea is the rule of representation: separate device-specific knowledge from the logic of PM context management. We design and prototype a *generic context manager* that provides device-independent suspend/resume logic while allowing device vendors to fold device-specific knowledge into data structures to drive the logic. Our evaluation on the BeagleBone Black platform shows that our design can move most of suspend/resume code out of drivers, and is as effective as the original driver-based PM context management on power savings.

Acknowledgments

First of all, I would like to express my sincere thanks to my advisor Dr. Lin Zhong, for his great support and guidance. Through three years of study and research with him, he has not only guided me to think deeply and critically, but also helped me to develop a better research taste. I also want to thank my thesis committee members, Dr. Joseph Cavallaro and Dr. Dan Wallach for giving me valuable feedback on my thesis.

I am grateful to have my group mate Min Hong Yun to help with my project, his contribution to the project helps a lot to the motivation chapter of this thesis. I also thank my former office mate Robert LiKamWa and current office mate Kevin Boos for their valuable feedback and suggestions in my daily research and life.

Moreover, I thank my other RECG group mates and all my friends for their continuous help and support.

Finally, I want to thank my parents for firmly supporting all the decisions I made. As rural farmers, they may never see or understand this thesis, but I want to let them know that I cannot accomplish this thesis without their care and love.

Contents

Abstract	ii
Acknowledgments	iii
List of Illustrations	vi
List of Tables	ix
1 Introduction	1
2 Background	5
2.1 Device and their power-saving modes	6
2.2 PM context management	8
2.3 Moving functions out of drivers	11
3 Motivations	14
3.1 Analysis methodology	14
3.2 Driver-based <code>suspend/resume</code> functions are expensive	17
3.2.1 <code>suspend/resume</code> functions account for significant SLoC count .	17
3.2.2 <code>suspend/resume</code> functions account for significant maintenance efforts	18
4 Device-independent Context Management	19
4.1 Design overview	19
4.2 Reentrancy	21
4.3 Generic <code>suspend/resume</code> logic	24
4.3.1 Generic logic contains common PM activities	24

4.3.2	Device-specific PM activities remain in device drivers:	28
4.4	Device-specific knowledge representation	30
5	Implementation	35
6	Evaluation	37
6.1	Evaluation setup	37
6.2	Effective power management	38
6.3	Reduced device driver complexity	41
6.4	Latency	44
6.5	Limitations	47
7	Related Work	49
7.1	Moving things out of driver	49
7.2	Representing device knowledge	50
7.3	Move drivers out of kernel	51
7.4	Driver fault tolerance and correctness	53
8	Concluding Remarks	54
	Bibliography	56

Illustrations

- 2.1 A simplified view of modern mobile application processors. Any module that is not a CPU or memory is considered as a device and managed by a device driver running on the CPU as part of the high-level operating system. A device can be on-chip, integrated inside the application processor, e.g., the I2C controller, or off-chip, e.g., the sensor connected to the I2C interface. On-chip devices are usually organized into hierarchical domains for power management. 6
- 2.2 Device driver plays two roles in power management: (1) track device usage by calling into the PM frameworks; and (2) provide callbacks (marked as grey) for the PM frameworks to suspend/resume the device. Linux kernel provides numerous device-independent frameworks to simplify driver development, including the PM frameworks themselves. Our goal is to move the `suspend/resume()` callbacks out of drivers and realize them as a device-independent framework. 7
- 3.1 Evolutionary history of average SLoC for 10 drivers that are common for Nexus 5/6/6P/5X since the release date of Nexus 5. The most recent devices have increased the average SLoC by more than 40% for the same drivers. 16

3.2	Accumulative number of commits for 10 drivers that are common for Nexus 5/6/6P/5X since the release date of Nexus 5. After the release date of Nexus 5, the suspend/resume functions for the same drivers are still under active development on newer versions of the Nexus devices before their releases.	16
4.1	Design overview of generic context manager. The generic context manager maintains per-device context containers for each device to privately contain their PM context. It provide generic suspend/resume functions and takes device-specific knowledge provided by device vendors to manage PM context for various devices.	20
4.2	Generic suspend/resume logic in the gray box. Runtime and system PM share the same set of common PM activities for suspending/resuming respectively (not all of them are shown in this figure). A subset of the common PM activities are executed for a device according to the device-specific knowledge. Device-specific PM activities remain in device drivers. The figure shows a default order for the PM activities, but device-specific knowledge can specify a different order if necessary.	25
4.3	Power management as a simplified finite-state machine. Runtime and system PM may have different power-saving states. In runtime PM, runtime suspend() brings a device from the functional state to the runtime power-saving state, and runtime resume() does the reversal. In system PM, system suspend() can bring a device into the system power-saving state from either the functional or the runtime power-saving state, and system resume() has to make sure the device is brought back to the same state as it was in before it was put into the system power-saving state.	28

6.1	Power rails on BBB, resources from [1, 2]. Not all the devices in the power domains are showed in the figure. When the system is suspended, VDD_MPU and VDD_CORE will be scaled down to 0.95V, all other rails remain on. This causes the overall power consumption to be 243 mW when the system is suspended.	43
-----	--	----

Tables

3.1	SLoC counts of suspend/resume functions on 4 Nexus phones. suspend/resume functions account for a significant portion of SLoC in device drivers.	17
5.1	Suspend/resume support for the five drivers modified in our implementation. ✓ callbacks provided; ✗: callbacks not provided. In our implementation all the 5 drivers use the generic suspend/resume functions as their PM callbacks.	36
6.1	Overall power consumption on BBB (mW). The generic context manager can achieve the same level of power savings as the original driver-based PM. When measuring the power for runtime PM, we disable/enable the runtime PM feature only for the devices supported by our modified drivers.	39
6.2	SLoC count for suspend/resume code that still remains in device drivers. Generic context manager can move most of the suspend/resume code out of the 5 drivers, as a result it reduces driver complexity.	42

- 6.3 Execution latency of **suspend/resume** functions in CPU cycles, we configure the CPU frequency to be 1GHz, so the numbers are also in nanoseconds. Generic **suspend/resume** functions could be 1x to 9x slower than the original ones. But the absolute execution time is very small. 45

Chapter 1

Introduction

Most of the code in Linux is device drivers, so most of the Linux power management (PM) code is also driver-specific. Most drivers will do very little; others, especially for platforms with small batteries (like cell phones), will do a lot. [3]

—Rafael Wysocki, Maintainer of Linux Device Power Management

In this work, we demonstrate that much of the PM suspend/resume code can be realized outside the notoriously complex device drivers. This simultaneously achieves two objectives: (1) simplify device drivers and their development, which is notoriously error-prone, and (2) improve system energy efficiency because many device drivers do not provide adequate support for power management [4].

With tight battery and thermal budgets, mobile systems must carefully manage their devices so that idle ones can stay in the lowest possible power mode. This problem is generally known as device *power management* (PM). *System* PM has been extensively studied and is widely used today. For example, Android aggressively suspends a system into a low-power mode after a brief period of user inactivity, unless an application holds a wakelock [5,6]. In contrast to system PM, *runtime* PM

is concerned with putting individual idle devices into a low-power mode even when the rest of the system is serving the user. In Android-based mobile systems, runtime PM is controlled by the Linux kernel while system PM by Android itself. Even when a wakelock prevents Android from suspending the system, the Linux kernel can still perform runtime PM of devices.

In today's systems, power management relies on device drivers for two critical roles. First, for runtime PM, device drivers are responsible for tracking the device usage, e.g., by invoking the APIs exposed by the Linux runtime PM framework. Second, the device driver implements the context saving and restoration functions necessary for power management. Before allowing a device to enter a low-power mode where its states may be lost, the system must save its states (or context); likewise, the system must restore its states when resurrecting the device from a low-power mode to be functional. The device drivers provide functions for both saving and restoration, i.e., `suspend()` and `resume()`, often separately for system and runtime PM.

While it is *convenient* to have the device drivers assume the responsibility of doing device power management, it also leads to three problems that are increasingly important. First, it adds to the complexity of drivers, which is already notoriously high. The suspend/resume functions contain low-level knowledge that is device-specific, tedious and error-prone. Second, related to the first problem, many device drivers today forgo their responsibility completely, providing inadequate implementation of `suspend/resume` functions. This requires continuous development efforts from driver

developers to maintain and update the power management code. Finally, because device drivers are executed from the powerful CPU, driver-based power management means that the CPU has to be always involved, limiting the effectiveness of power management, especially runtime PM. See §2 for details.

A recent work [4] has showed that it is possible to relieve the first role of device drivers in power management but still relies on the drivers to supply the **suspend/resume** functions. This work, however, focuses on relieving the second role by device drivers in power management. That is, move **suspend/resume** functions out of drivers and make them generic to devices. This goal is considerably harder and has a wider system impact. First, the key idea of [4] was to infer whether a device has pending tasks outside of driver. That is, it bypasses the drivers by monitoring access to device registers or polling a specially designed status register bit exported by the device hardware. Because on-chip device registers in ARM systems are usually memory-mapped, one only needs to know which memory addresses to monitor or poll. In contrast, **suspend/resume** functions involves intimate interaction between driver and hardware and requires much more device-specific knowledge as will be elaborated in §2. In order to move **suspend/resume** functions out of drivers, our key idea is to aggressively apply the famed *Rule of Representation*, which seeks to “fold knowledge into data so that logic can be simple and robust” [7]. Second, while the first role targeted by [4] is only involved in runtime PM, the second role targeted by this work is critical for both system and runtime PM. Although runtime and system **suspend/resume** functions are

usually implemented separately in the same driver, our approach will work for both. As a result, our work will not only complement that of [4] to provide runtime PM to devices but also enable system PM for devices whose drivers do not support it.

In this thesis, we present *generic context manager*, a novel design that moves **suspend/resume** functions out of device drivers. The generic context manager provides generic **suspend/resume** functions for both runtime and system PM for various devices. It takes device-specific PM knowledge in data structures to manage their PM context. We implement the generic context manager as a centralized kernel module in Linux 4.1 on the BeagleBone Black development platform. Our experimental evaluation shows that the generic context manager not only can achieve the same level of power savings as the original driver-based power management, but also moves most of the suspend/resume code out of device drivers.

In summary, this thesis makes the following two contributions:

1. We present a novel design of generic context manager that provides generic suspend/resume logic for various devices and manages device PM context according to device-specific knowledge in data structures.
2. We present a prototype implementation of the generic context manager. We experimentally show that it achieves the same level of power savings as the original driver-based power management and reduces driver complexity by moving most of the suspend/resume code out of drivers.

Chapter 2

Background

The first principle of energy efficiency is frugality. That is, unused, idle components of a computer system should enter power-saving (PS) mode, or *power managed*. If the rest of the system is still functional, the component is said to be runtime power managed. When the entire system enters a non-functional power-saving mode, it is said to be system power managed. In this case, the system often saves the states of its components into the memory, e.g. suspend to RAM, or the non-volatile storage, e.g., suspend to disk.

Power management is not only critical to the battery lifetime of mobile systems but also important to meet the thermal and peak power constraints facing modern computers: fundamentally idle components should remain *dark*, to use the popular jargon of dark silicon [8].

We next describe how Linux performs device power management, both *runtime* and *system*. In particular, we focus on the roles played by device drivers. Other OSes like Mac OS X [9] and Windows [10] have drivers play similar roles in device power management.

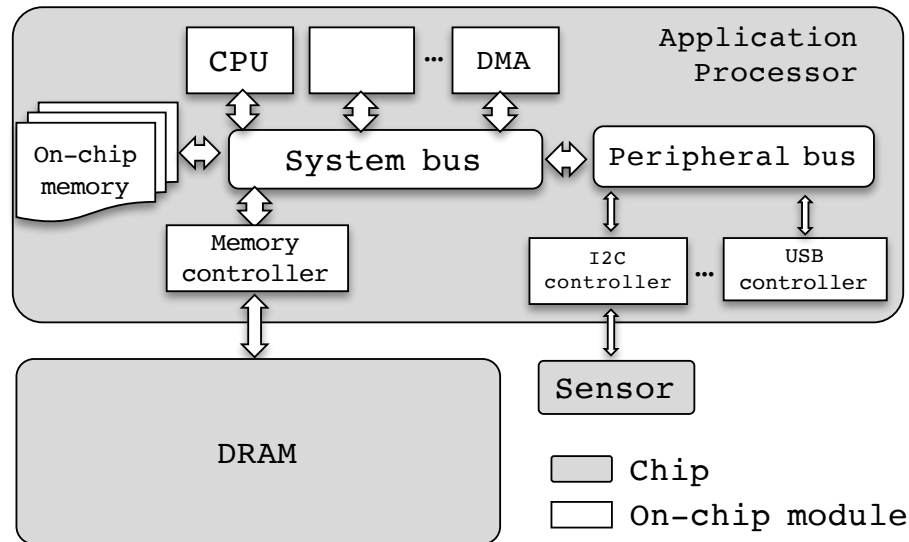


Figure 2.1 : A simplified view of modern mobile application processors. Any module that is not a CPU or memory is considered as a device and managed by a device driver running on the CPU as part of the high-level operating system. A device can be on-chip, integrated inside the application processor, e.g., the I2C controller, or off-chip, e.g., the sensor connected to the I2C interface. On-chip devices are usually organized into hierarchical domains for power management.

2.1 Device and their power-saving modes

We first provide the necessary hardware background. Figure 2.1 presents a highly simplified view of the hardware of mobile systems. It includes many chips, the application processor, main memory (DRAM), various sensors and other I/O devices, and wireless interface cards (not shown). The application processor is a system-on-a-chip (SoC) with the actual multi-core CPU being only a relatively small fraction of the chip. The majority of the chip is occupied by numerous modules, including hardware accelerators such as video codecs, DSPs, and GPUs, I/O controllers and DMA. The CPU and the modules are integrated with a hierarchy of buses, often a system bus and a peripheral bus, as shown in Figure 2.1. On the chip, one will also

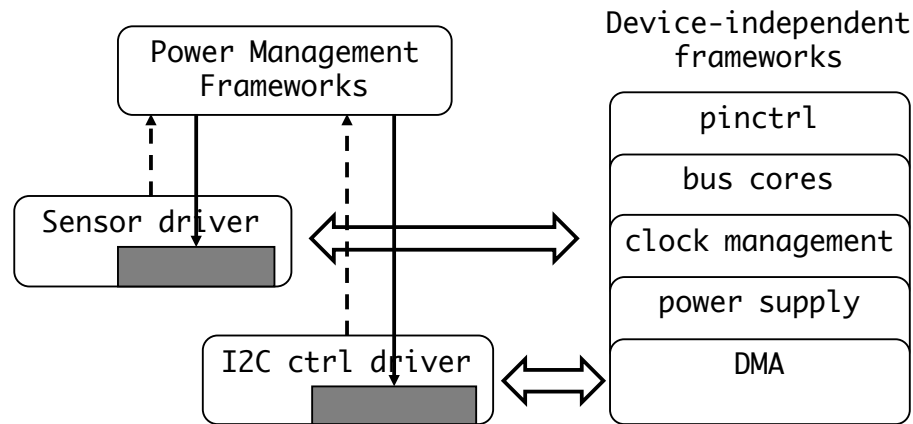


Figure 2.2 : Device driver plays two roles in power management: (1) track device usage by calling into the PM frameworks; and (2) provide callbacks (marked as grey) for the PM frameworks to suspend/resume the device. Linux kernel provides numerous device-independent frameworks to simplify driver development, including the PM frameworks themselves. Our goal is to move the `suspend/resume()` callbacks out of drivers and realize them as a device-independent framework.

find other special hardware such as small ROM and RAM that are mapped into the memory view of the CPU. Linux, as a high-level OS, runs its software on the CPUs; To them, the other on-chip modules are devices that are managed by the OS with *device drivers*.

To conserve power, all hardware components except a very few on-chip devices implements power-saving modes. A device driver plays two critical roles for the corresponding device to enter a power-saving mode, as shown in Figure 2.2. First, for runtime PM, the device driver determines if a device is idle. It does so by tracking the device usage, e.g., by invoking the APIs provided by the Linux runtime PM framework. Only an idle device should ever enter a power-saving mode. Second, the device driver must make it *safe* for an idle device to enter a power-saving mode in which the device may not be able to retain all the states. This is done by calling a

`suspend()` function. Likewise, the driver must make the device *ready* for service when it wakes up from a power-saving mode. This is done by calling a `resume()` function.

To amortize the hardware overhead of power management, multiple on-chip devices can share the same clock source and its management circuitry and they constitute a *clock domain*. Multiple clock domains can share the same on-chip power source and its management circuitry to form a *power domain*. The hardware of a domain is also considered a device and is managed by its device driver. Necessarily, before a domain enters a power-saving mode, it must be safe for all its members to enter power-saving modes.

2.2 PM context management

The `suspend/resume` functions provided by drivers manage the *PM context* of devices. When `suspend()` readies a device for power-saving modes, it must save the PM context; Likewise, when `resume()` resumes a device for service, it must restore the PM context.

The PM context is best explained by its four components: execution, interface, wakeup, and physical. First, a digital component can be viewed as a finite state machine, and the *execution context* is the state it is currently in. A device's state information is encoded by its storage elements, often including registers and memories. Ideally when a device gets out of a power-saving mode, it should get back to the same state as it was in before it gets into the power-saving mode. We note that an

important part of the execution context is the software-programmable configuration, e.g., enabled features (e.g., interrupt) and operational settings (e.g., baud rate for UART controller). Not all drivers have to deal with execution context. Many devices have internal mechanisms to save their execution context. For example, on-chip DSP, GPU, and micro-controller cores run their own software and have access to either the main memory or their internal memory that retains content in power-saving modes; some off-chip devices may also be able to save their own execution context. On the other hand, most on-chip devices and many off-chip devices do rely on their drivers to save and restore their execution context. As a result, they must make their execution context accessible to software running on the CPU, i.e., the driver, in the form of registers. On ARM-based SoCs, registers for on-chip devices are memory-mapped. That is, software can access them as if they are part of the memory. For off-chip devices like the sensor in Figure 2.1, software can access their registers via the necessary buses, e.g., I2C in the sensor example.

Second, a device interfaces with the rest of the system and the world; two important interfaces are stateful: direct memory access (DMA) and interrupt. We call such states the *interface context*. Instead of saving the interface context, `suspend()` makes sure it vanishes properly; `resume()` then simply readies the interfaces. For DMA, the Linux DMA framework provides device drivers with APIs to synchronize data transfers on the DMA channels to prevent data loss before the devices are suspended. Thus the `suspend/resume` functions utilize these APIs to properly stop/start

DMA channels. Alternatively when there is data on a DMA channel, `suspend()` can copy and save the data and `resume()` will resubmit the data for DMA. Additionally, `suspend()` will check if there is any pending interrupt to the device because an interrupt may occur after the system considers the device is idle. When there is an interrupt, the driver can either delay suspending using the *autosuspend* mechanism or simply gives up suspending.

Third, `suspend()` must configure the device properly for wakeup, i.e., the conditions under which the device should leave a low-power mode. We call this configuration the *wakeup* context.

Finally, the system provides a physical context for its devices. It supplies clock signal and power. The `suspend/resume` functions must properly disengage a device from and engage it with this context, respectively. Usually the PM framework will invoke the power domain's `suspend/resume` functions to disable/enable necessary clocks and power for a device, but device drivers have to deal with the rest part of the physical context that is not handled by the PM framework. For example, if a GPIO controller has an enabled debounce clock, its driver has to disable/enable this clock properly in the `suspend/resume` functions. For clock management, device drivers invoke APIs from the clock framework to disable/enable their clocks; The clock framework will keep track of the clocks' usage and gate them properly.

Runtime PM vs. System PM: Runtime and system PM's `suspend/resume()` functions are similar but have important differences. First, the implementations of

runtime and system `suspend/resume` functions are based on different assumptions. For runtime PM, when runtime `suspend()` is called, it assumes the device is in quiescent state and no one is using the device; when there is new use on the device, runtime `resume()` will resume the device to be functional. For system PM, system `suspend()` assumes the device is always in use, it has to do whatever necessary to bring the device to a power-saving mode like suspend-to-RAM; when the system is resumed, system `resume()` will resume the device to the state it was in before it is put into the power-saving mode. That is to say, even if a device is previously in runtime suspended mode when its system suspend is triggered, the device has to be put into the same runtime suspended mode when the system is resumed.

Moreover, the timings to invoke `suspend/resume()` functions are different. On one hand, runtime PM is synchronous regarding device activities. For example, runtime `suspend()` is called when there is no pending activity for a device. On the other hand, system PM is asynchronous. A user can press the power button any time to suspend the system, even when some devices are still processing tasks. Thus before calling system `suspend()` to manage device PM context the PM framework has to properly freeze the running tasks.

2.3 Moving functions out of drivers

Device drivers are notorious for accounting for a large share of the kernel code and an even larger share of the kernel bugs and security exploits. The reasons are well-

known: they are difficult to develop and require intimate hardware knowledge and using unsafe programming languages, i.e., C and assembly. And they come from hardware vendors of varying competence.

Making device drivers more reliable and more secure has been a lasting research topic over the past two decades, including bug finding [11,12], fault tolerance [13–15], new architecture [16,17], new programming languages [18–20], and even automatic synthesis [21,22]. The one strategy that has seen actual adoption is nevertheless very simple: moving things out of drivers. The Linux kernel today provides numerous device-independent frameworks such as *pinctrl* [23] and *clock* [24] to provide functions necessary for device operations as shown in Figure 2.2. For example, the runtime PM framework provides a reference counting mechanism for a driver to track its device usage [25]. This strategy not only reduces the driver development effort but also makes it easier to guarantee drivers’ correctness. For example, instead of auditing each every driver’s reference counting, the Linux kernel only needs to audit its runtime PM framework. This strategy obviously contributes to the decreasing share by drivers in the Linux kernel code, from 70% to 57%, and the decreasing fault density in drivers, despite the growing diversity of Linux-ready hardware devices, according to studies carried out a decade a part [11,12].

Despite the success of the “moving-it-out” strategy, the function of power management has remained in the device drivers. It is not difficult to understand why: the diversity of hardware devices and the required interaction between hardware and

software for power management just make it extremely hard to realize it in a device-independent manner. Recently, the authors of [4], motivated by that many device drivers do not properly use the reference counting mechanism provided by the runtime PM framework, showed that tracking device usage can be done without the driver. This largely relieves device drivers from the first of the two roles in power management. However, the harder role of PM context management still remains in the driver. That is, the `suspend/resume` functions are still device-specific. The goal of this work is to move PM context management out of driver and into a device-independent framework.

In the rest of the thesis, we will first quantify the development effort of PM context management in Linux, i.e., device-specific `suspend/resume` functions, and then present our design and implementation of device-independent PM context management.

Chapter 3

Motivations

In this chapter, we show that it is highly desirable to move the PM context management code, i.e., the `suspend/resume` functions, out of device drivers. We build a tool called *ddAnalyzer* to count the source lines of code (SLoC) of `suspend/resume` functions in device drivers for four most recent Nexus phones. We also analyze the evolutionary history of the `suspend/resume` functions for those phones. Our analysis shows the `suspend/resume` functions contribute to a significant portion of code in drivers and require significant maintenance efforts from driver developers.

3.1 Analysis methodology

We clone the kernels from the vendors' repositories for the four Nexus phones, i.e., Nexus 5/6/6P/5x. We count the SLoC of the `suspend/resume` functions for each device driver using the latest version of the source code. All the four Nexus phones use device trees [26] to describe their hardware devices. Each phone has a device tree source (`.dts`) file that comes with the kernel source. The `.dts` file usually includes several `.dtsi` files that describe common hardware for multiple platforms, together they describe the devices used on the phone. To analyze the evolutionary history of

the `suspend/resume` functions on a phone, our analysis starts from the `.dts` file of that phone.

First, *ddAnalyzer* takes the `.dts` file of a phone as input and searches the corresponding drivers for devices described in the `.dts` file and the `.dtsi` files it includes in the kernel source. Then it traces each driver to check if the driver provides any `suspend/resume` functions for either runtime PM or system PM. Note that we only trace drivers that are configured to be compiled in the kernel `.config` file. Second, once *ddAnalyzer* finds a `suspend()` or `resume()` function in a driver file, it searches the commit history of the driver file for any modifications on the function and the functions it calls. The analyzer then generates an evolutionary history for the function based on all the modifications found. Third, for the latest version of `suspend/resume` functions *ddAnalyzer* finds in the history, it generates the SLoC of the function using `cloc` [27]. We count the SLoC of a function within the scope of the driver file. That is to say, *ddAnalyzer* builds a function calling tree starting from the found `suspend()` or `resume()` function (root caller) within the driver file. It adds the SLoCs of the callees in the tree to the SLoC of the root caller. But the SLoC of a callee is only counted once even if it could be called multiple times.

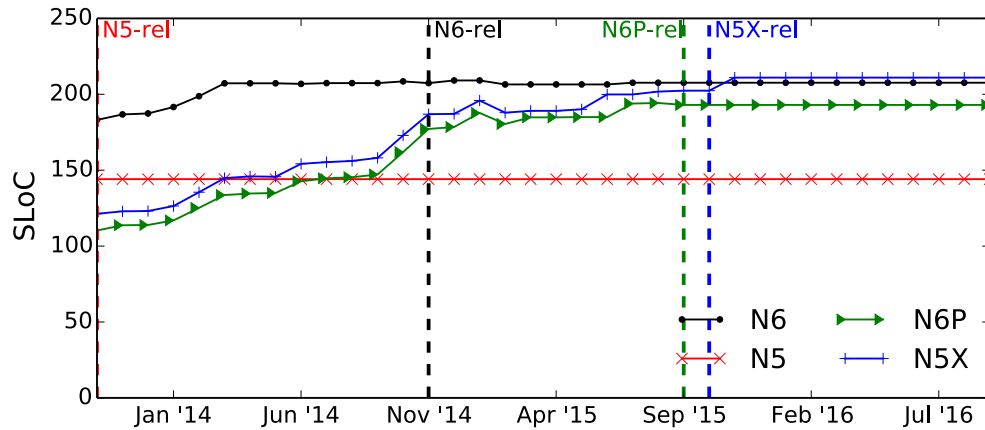


Figure 3.1 : Evolutionary history of average SLoC for 10 drivers that are common for Nexus 5/6/6P/5X since the release date of Nexus 5. The most recent devices have increased the average SLoC by more than 40% for the same drivers.

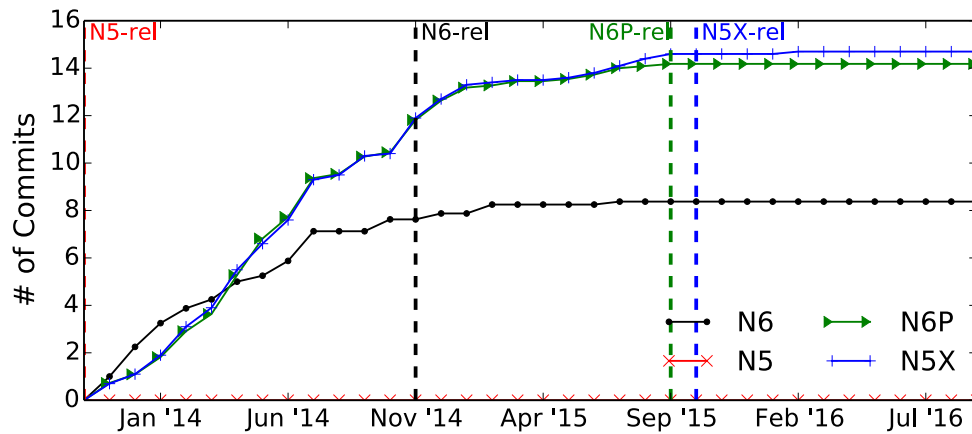


Figure 3.2 : Accumulative number of commits for 10 drivers that are common for Nexus 5/6/6P/5X since the release date of Nexus 5. After the release date of Nexus 5, the **suspend/resume** functions for the same drivers are still under active development on newer versions of the Nexus devices before their releases.

Mobile devices	SLoC per driver	Number of drivers	Total SLoC
Nexus 5	117.5	19	2233
Nexus 6	218.6	17	3717
Nexus 6P	226.3	20	4526
Nexus 5X	237.7	19	4517

Table 3.1 : SLoC counts of `suspend/resume` functions on 4 Nexus phones. `suspend/resume` functions account for a significant portion of SLoC in device drivers.

3.2 Driver-based `suspend/resume` functions are expensive

3.2.1 `suspend/resume` functions account for significant SLoC count

Table 3.1 shows the SLoC of `suspend/resume` functions in device drivers for 4 latest Nexus phones. On average there are more than 220 lines of `suspend/resume` code per driver on the most recent Nexus 5X and Nexus 6P phones (second column in Table 3.1), which contributes to over 4000 SLoC in total for all `suspend/resume` related functions on those devices. Even though those numbers are not considerably big compared to the total number of SLoC of Linux kernel, they still account for a significant portion of device driver code base. Our analysis shows that the SLoC of `suspend/resume` functions accounts for 7.4% to 10.5% of all device drivers within the same files we analyze (third column in Table 3.1). We conclude that `suspend/resume` functions actually add significant complexity to device drivers which have already been highly complex and error-prone.

3.2.2 `suspend/resume` functions account for significant maintenance efforts

We also analyze the evolutionary history of the above four Nexus mobile devices. All these devices use the same series of Snapdragon 800 SoCs, we find 10 drivers that are common for them. Figure 3.1 and Figure 3.2 shows the SLoC and number of commits evolution for the 10 drivers on the four devices. Figure 3.1 shows that since the release date of Nexus 5, the most recent devices (Nexus 6P and Nexus 5X) have increased the average SLoC by more than 40% for the same drivers. Figure 3.2 shows that after the release date of Nexus 5, the `suspend/resume` functions for the same 10 drivers are still under active development on newer versions of the Nexus devices before their releases. The added SLoC are mainly for fixing bugs and adding new features. For example, the above four phones share the same SPI controller driver. The runtime PM framework could view unbalanced clocks on the SPI controller when the SPI driver is unable to propagate local resource getting errors to the runtime PM framework in its runtime `resume()` function. This bug was fixed on Nexus 6P [28]. Those two figures indicate that `suspend/resume` functions constantly needs significant maintenance efforts from driver developers.

Therefore, our work aims for moving `suspend/resume` functions out of device drivers to reduce PM code size and simplify driver development.

Chapter 4

Device-independent Context Management

In this chapter, we present our novel design of a *generic context manager* as shown in Figure 4.1 to move `suspend/resume` functions out of device drivers. The generic context manager is a device-independent framework that provides generic `suspend/resume` functions to manage both runtime and system PM context.

4.1 Design overview

Our key idea to achieve device-independence is to separate the logic from the device-specific knowledge that are used to suspend/resume devices. It is feasible to implement `suspend/resume` functions with generic logic for various devices while folding device-specific PM knowledge into data structures, guided by the *rule of representation* [7]. For device-independent PM context management, the generic context manager has to resolve three important technical challenges.

1. The generic context manager provides generic `suspend/resume` functions for multiple devices, thus it has to support reentrancy to allow possible concurrent invocations of the `suspend/resume` functions.
2. We have to decide what PM activities should be made device-independent.

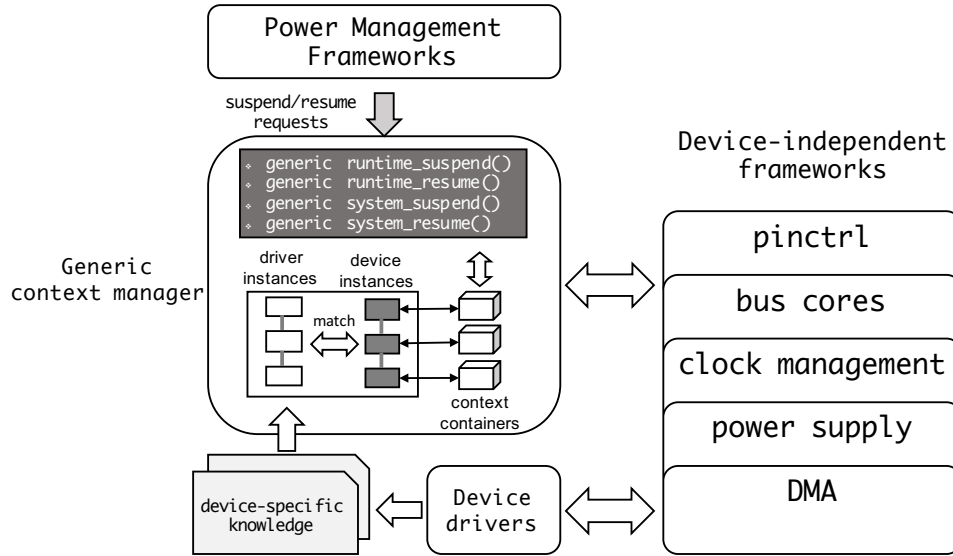


Figure 4.1 : Design overview of generic context manager. The generic context manager maintains per-device context containers for each device to privately contain their PM context. It provide generic **suspend/resume** functions and takes device-specific knowledge provided by device vendors to manage PM context for various devices.

3. We must design data structures to concisely represent device-specific PM knowledge, while providing adequate device-specific PM information to the generic logic.

Figure 4.1 shows the design overview of the generic context manager by putting it into the same scope as shown in Figure 2.2. To resolve the above three challenges, the generic context manager design contains three major components, i.e., context containers associated with device and driver instances, generic **suspend/resume** functions and device-specific knowledge representation. The context container resolves the reentrancy challenge by maintaining the devices' PM context privately to them. The generic **suspend/resume** functions provides suspend/resume callback interfaces to

the PM framework, internally they contain the generic suspend/resume logic. And the device-specific knowledge is represented by data structures such as register tables and flags, which are passed to the generic context manager to drive the generic logic. The generic context manager accesses devices using the same set of APIs from the device-independent frameworks as what the original device drivers do, such as using `read/write()` to access memory-mapped registers and `clk_disable/enable()` to disable/enable clocks.

In the following, we describe how we resolve the above three challenges by the three components in detail.

4.2 Reentrancy

The generic context manager is aiming to manage PM context for multiple devices, it is possible that there are concurrent invocations of the `suspend/resume` functions; and one invocation should not influence another. This naturally leads to the requirement that the generic `suspend/resume` functions should be reentrant. To design reentrant `suspend/resume` functions, we have to meet three requirements [29]:

- The `suspend/resume` functions should not use global non-constant or static variables;
- The `suspend/resume` functions should not call non-reentrant functions;
- The `suspend/resume` functions should not modify their own code.

Because the `suspend/resume` functions will not modify their code, the third requirement is automatically met. In the following, we focus on elaborating how we meet the first and the second requirements in our generic context manager design.

No global non-constant or static variables: Since the `suspend/resume` functions should not use global non-constant or static variables, the generic context manager has to keep device-specific PM context private to individual devices. As a result, it has to create a clean isolation among the PM context of different devices. To isolate PM context of different devices, we design a per-device PM *context container* to contain the PM context of each device privately, following the device driver “state container” design pattern [30]. One invocation of the `suspend/resume` functions is only allowed to access the context container of the device that is being power managed, it cannot access other devices’ context containers.

A context container is associated with a device through an device instance registered by the device in the generic context manager, as shown in Figure 4.1. Linux kernel uses `struct device` and `struct device_driver` to present a device and a driver respectively. To use the generic `suspend/resume` functions for a device, both the device and its driver have to register an instance in the generic context manager. The device instance is responsible for providing per-device knowledge and hooking its corresponding context container, while the driver instance is responsible for providing device-specific knowledge for all the devices that the driver supports. The device instance and the driver instance are linked via a `match` function so that a device’s

context container can easily refer to the its driver-provided knowledge. This linking design has at least two benefits: 1) It allows one instance of driver-provided knowledge to serve multiple devices of the same type, which is commonly seen in modern operating systems; 2) It also supports per-device suspend/resume behavior tuning. Individual devices can provide additional knowledge through their device instances to the generic context manager to tune a PM activity only for themselves, such as choosing a locking mechanism type.

A context container contains the PM context for each device, besides, it also contains necessary information to access the device. For example, the **suspend/resume** functions need to know the memory-mapped base address for device registers to access a device. The base address is configured and saved in the context container of the device during device initialization. Other device access information includes DMA channels, clock references and so on.

No non-reentrant function calls: We take the benefits of existing reentrancy support in the device-independent frameworks to meet the second requirement. Besides calling the reentrant functions created by our design, the **suspend/resume** functions also call functions from the device-independent frameworks such as *pinctrl* and *clock* frameworks that are designed for multiple devices, thus they should also be reentrant. As a result, the **suspend/resume** functions do not call non-reentrant functions.

However, there are still non-reentrant function calls in the generic context manager, such as memory allocation for the context container and register address remapping for memory-mapped devices. Those functions will not be called by the **suspend/resume** functions. They are called by a one-time initialization function that is called when a device instance is registered to the generic context manager. The device instance is registered after a device is added to the system but before it is probed.

4.3 Generic suspend/resume logic

To resolve the second challenge, the generic suspend/resume logic contains PM activities that are common across various devices, and it keeps the device-specific PM activities remaining in device drivers, as shown in Figure 4.2. The order of common PM activities in the generic logic can differ for different devices, we build a default PM activity order and allow device-specific PM knowledge to specify specific orders to execute the PM activities if necessary.

4.3.1 Generic logic contains common PM activities

When a device is suspended, it usually involves saving its execution and configuration context and disengaging it from interface and physical context (as discussed in §2). Each of these actions is called a PM activity. As shown in Figure 4.2, the generic suspend logic contains a sequence of PM activities, i.e., save execution context, pause DMA channels, disable IRQ, disable clock and select sleep pin state. These activities

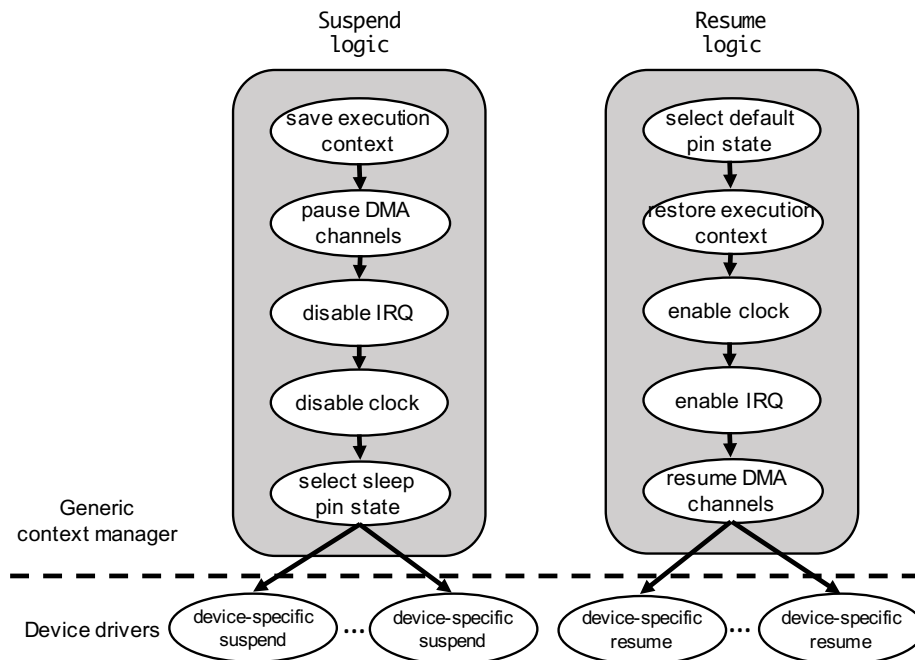


Figure 4.2 : Generic suspend/resume logic in the gray box. Runtime and system PM share the same set of common PM activities for suspending/resuming respectively (not all of them are shown in this figure). A subset of the common PM activities are executed for a device according to the device-specific knowledge. Device-specific PM activities remain in device drivers. The figure shows a default order for the PM activities, but device-specific knowledge can specify a different order if necessary.

are *common* across various devices when they are suspended, but not all the devices have to execute all of them. Each PM activity could involve reading/writing a set of device registers that are specified by device-specific knowledge (explained later). It is the device-specific knowledge that decides which activities should be executed for a device. The generic context manager relies on the device-independent frameworks to execute PM activities. For example, DMA channels can be paused with the channel pausing APIs from the DMA framework. In case any activity returns an error indicating that the managed device is busy, the whole generic suspend function should abort. The generic resume logic is a reversed process of suspend logic, i.e., basically restoring the execution and configuration context and engaging the device with its interface and physical context again. It is also the device-specific knowledge that determines what PM activities to execute in the resume logic for a device.

Different PM activity orders in the generic logic could result in different PM behaviors on devices. We build a default PM activity order as shown in the grey boxes Figure 4.2, but device-specific knowledge is allowed to specify their own PM activity orders in the generic suspend/resume logic. For example, one device has to disable clock for before disabling the IRQs while another device needs to disable IRQ before disabling clock. Our implementation in §5 uses the default PM activity and it works well to suspend/resume all the devices we consider.

Runtime PM logic vs. System PM logic: Runtime `suspend/resume()` and system `suspend/resume()` are functionally the same, they share the same set of

common PM activities as shown in Figure 4.2. Actually system `suspend/resume()` can reuse the code of the same PM activities in the runtime `suspend/resume()`. For example, system `suspend()` can reuse the code to save device's execution context from runtime `suspend()`.

However, runtime PM and system PM can have different power-saving modes when the devices are suspended, thus we have to separate their suspend/resume logic necessarily. Figure 4.3 shows device power management as simplified finite-state machine (FSM) when doing both runtime and system PM. As we discuss in §2, runtime `suspend/resume` functions are synchronous, runtime `suspend()` is called to bring a device from the functional state to the runtime power-saving state when it is not used; and runtime `resume()` is called to bring the devices back to the functional state when it is used again. On the other hand, system `suspend/resume` functions are asynchronous. System `suspend()` can be called at any time when a device is either in the functional state or in the runtime power-saving state. But no matter what state the device was in, the system `resume()` has to make sure when the device is waken up it will be put into the same state as it was in before. Since the generic context manager does not know which state a device could be in when system `suspend()` is called, it has to assume that the device could be in runtime power-saving state. Therefore, if the device is currently in the runtime power-saving state, the generic system suspend logic has to bring the device back to the functional state before it executes any PM activities to put the device into the system power-saving state. Vice

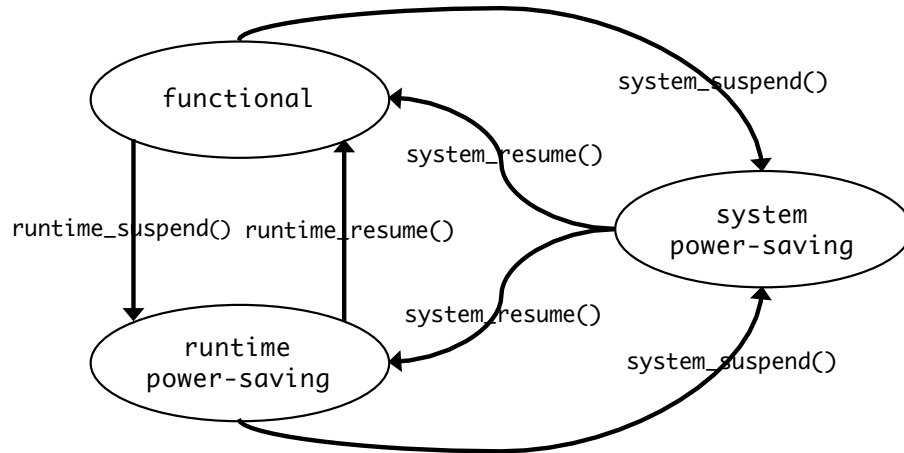


Figure 4.3 : Power management as a simplified finite-state machine. Runtime and system PM may have different power-saving states. In runtime PM, runtime `suspend()` brings a device from the functional state to the runtime power-saving state, and runtime `resume()` does the reversal. In system PM, system `suspend()` can bring a device into the system power-saving state from either the functional or the runtime power-saving state, and system `resume()` has to make sure the device is brought back to the same state as it was in before it was put into the system power-saving state.

versa, the generic system resume logic has to put the device into runtime power-saving mode after it executes necessary PM activities to put the device into the functional state. Tracking the runtime PM state of a device can be easily done by calling APIs from the runtime PM framework.

4.3.2 Device-specific PM activities remain in device drivers:

Although we can as much code in `suspend/resume()` as possible out of device drivers, we find some *device-specific* power management code have to remain in the drivers to execute device-specific PM activities as shown in Figure 4.2. Those activities include: a) dealing with device silicon bugs; b) setting up interactions with device-specific frameworks; c) configuring device-specific remote wakeup.

Device hardware can contain silicon flaws, and device drivers have to explicitly handle those flaws to make the devices functional. For example, the MMC host controller on TI OMAP5 can report false timeout command under high-speed mode [31, 32]. This errata has to be handled in the MMC controller driver's runtime `resume()` function when setting up clocks. PM activities like this cannot be moved out from device drivers. Also, functions in device drivers usually are not called directly by device users, instead they are registered with upper layer frameworks which provide standard APIs to the device users. For example, an 8250 UART driver has to register an 8250 port interface during device probe, after which all the device accesses are done through the port interface [33]. We find it hard to separate suspend/resume logic from the device-specific frameworks when device drivers are tightly coupled with them, unless the driver stack is fully redesigned. Thus in our design we also leave this kind of PM activities in the device drivers. Finally, runtime PM requires devices to be remote wakeup-capable, but some devices have specific wakeup configurations. For example, level-detectable GPIO pins are configured to be edge-detectable during runtime suspend such that they can generate wakeup events to the PRCM [34]. If we can design the hardware in a way that all the wakeup configurations are done using a standard, such as writing a series of registers, we can still move this part of code to the generic logic.

4.4 Device-specific knowledge representation

The suspend/resume logic we describe above is generic, it only follows the device-specific knowledge represented by data structures to manage device PM context. How to represent the device-specific knowledge is the key to reduce driver development effort and make device drivers less error-prone. We use tables and status flags to represent device knowledge, which is simple enough to relieve driver developers' effort to write complex code and also provides adequate information for the generic suspend/resume logic to manage devices' PM context.

Device knowledge representation has two parts: (1) Vendors are supposed to provide device-specific knowledge in tables and flags; and (2) Operating system kernel saves the tables and flags in memory and associates them with the corresponding devices and drivers.

The first part is not complicated, once the specifications of the tables and flags are well defined, vendors can generate device knowledge tables and flags from the device technical reference manual (TRM). We can extend existing solutions using domain-specific languages like Devil [19] and HAIL [20] to automatically generate the tables and flags. Those tables and flags can be passed to the kernel by compiling them into the kernel image or other binaries such as device tree blob. We will focus on discussing the second part.


```

1      static u32
2      omap_i2c_reg_context[] = {
3          [OMAP_I2C_IE] = 0,
4          [OMAP_I2C_CON] = 0,
5          [OMAP_I2C_PSC] = 0,
6          [OMAP_I2C_SCLL] = 0,
7          [OMAP_I2C_SCLH] = 0,
8          [OMAP_I2C_IRQENABLE] = 0,
9          [OMAP_I2C_WE] = 0,
10     };
11

```

Listing 4.1: I2C controller's context array representation

```

1      static universal_reg_entry
2      omap_i2c_save_context_tbl[] = {
3          {
4              .reg_op = PM_REG_READ,
5              .reg_offset = OMAP_I2C_PSC_REG,
6              .ctx_index = OMAP_I2C_PSC,
7          },
8          {
9              .reg_op = PM_REG_READ,
10             .reg_offset = OMAP_I2C_SCLL_REG,
11             .ctx_index = OMAP_I2C_SCLL,
12         },
13         ...
14     };
15

```

Device knowledge representation in kernel: Device-specific knowledge is represented in two categories in the kernel, *per-driver knowledge* and *per-device knowledge*. As we indicate above, it is very common for a driver to support multiple devices. Thus *per-driver knowledge* refers to the knowledge that is common across all devices supported by the same driver; *per-device* knowledge refers to the knowledge of a single device that has individual requirements regarding doing power management. In the perspective of generic context manager, per-driver knowledge is provided through the driver instances while per-device knowledge is provided through the device instances. As we discuss above in §4.2, the per-device context container contains the reference of per-device knowledge because it is directly hooked to the device instance. Once a driver instance and a device instance are matched by the `match` function, the device's context container will save the reference to the per-driver knowledge and access it easily.

Per-driver knowledge includes a context array that is of interest to doing power management and register tables for suspend/resume PM activities in the generic logic. The context array is provided by a device driver for all the devices it supports. When a device is probed, the generic context manager will create a copy of the context array and save its reference to the context container of the device. All the PM activities read/write a device's registers based on its context array. For each PM activity, the involved registers are represented in one or more tables. Each table entry indicates an access operation (read/write), the values and the index in the context

array for a device register. For example, Listing 4.1 shows the context array for OMAP I2C controllers, and Listing 4.2 is a register table that is passed to the *save execution context* PM activity in the generic suspend logic. The two entries shown in the *save context table* indicate that the PSC register and the SCL low register on an I2C controller are saved to the PSC and SCLL entries in its context array respectively. The register tables can also embed simple logic to help simplify the generic suspend/resume logic. For example, device vendors can provide two register tables to the *restore execution context* activity of the generic resume logic, i.e., a *restore context table* and a *check context loss table*. Before the resume logic restores device execution context from the first table, it can check if there is context loss from the second one. If there is no context loss, then no context restoring is needed for this activity.

Per-device knowledge is usually represented by status flags. For example, whether a device supports DMA can be indicated by a flag passed from the device instance, which will determine if DMA channels need to be paused/resumed in the generic suspend/resume logic. Per-device knowledge is directly associated with the device instance in the kernel, thus it is easy to use the status flags to tune the suspend/resume behaviors independently for individual devices.

Note that using tables and flags is not the only way to represent device-specific PM knowledge in the kernel. We use them to show the feasibility of folding device-

specific knowledge into data structures. Using other data structures to represent device knowledge requires further exploration.

Chapter 5

Implementation

Based on our design in §4 we implement the generic context manager as a centralized kernel module in Linux kernel 4.1 on the BeagleBone Black (BBB) development platform. BBB embeds an *AM3358* SoC that integrates a 1GHz Cortex-A8 processor and tens of on-chip peripherals. It uses a device tree to describe the hardware components on the board, which is compiled into a *device tree blob* and passed to kernel at booting time. We choose this platform because it is well-documented and its kernel source code is under active development by a large group of developers.

We modify 5 device drivers for BBB to register our generic suspend/resume functions as their runtime PM and system PM callbacks, including the I2C controller driver, the GPIO controller driver, the multimedia card (MMC) controller driver, the SPI controller driver and the LCD controller driver. Table 5.1 lists the suspend/resume support provided by the above 5 drivers, not all of them provide all the `suspend/resume` functions on BBB. We choose these drivers not only because they are widely used but also because they provide callbacks for either runtime `suspend/resume()` or system `suspend/resume()`, which can be used to best guide our design and evaluation of our generic `suspend/resume` functions. Each of these drivers can support one or more devices on BBB. Specifically, there are in total 3 I2C

Device drivers	original callbacks		generic callbacks	
	runtime	system	runtime	system
I2C	✓	✗	✓	✓
GPIO	✓	✗		
MMC	✓	✓		
SPI	suspend ✗ resume ✓	✓		
LCDC	✗	✓		

Table 5.1 : Suspend/resume support for the five drivers modified in our implementation. ✓ callbacks provided; ✗: callbacks not provided. In our implementation all the 5 drivers use the generic `suspend/resume` functions as their PM callbacks.

controllers, 4 GPIO controllers, 3 MMC controllers, 2 SPI controllers and 1 LCD controller that can be supported by those 5 drivers respectively. In our implementation, the generic context manager manages PM context of these devices for both runtime PM and system PM.

Chapter 6

Evaluation

In the evaluation, we experimentally answer two questions:

1. Is the generic context manager as effective as the driver-based `suspend/resume` functions?
2. How much does the generic context manager reduce the driver complexity?

To evaluate the PM effectiveness of generic context manager, we measure the overall power consumption of the whole BBB platform when `suspend/resume` functions are provided by generic context manager and the original device drivers respectively. We show that generic context manager can achieve the same level of power savings as the original driver-based `suspend/resume` functions when doing both runtime PM and system PM. We also count the SLoC of the `suspend/resume` functions in device drivers before and after we move them out, showing that we can either completely move or move at least 40% of the driver-based `suspend/resume` code out of device drivers.

6.1 Evaluation setup

To set up the evaluation, we connect BBB with a Monsoon Power Monitor [35] and power the BBB through a USB cable. To test the PM effectiveness on the LCD

controller, we connect the board with a micro HDMI cable to a G245H Acer HDMI monitor running at 1024×768 resolution. The LCD controller interface is converted to HDMI by an on-board HDMI framer TDA19988BHN that is connected on the I2C0 bus (Figure 6.1) of the SoC. For debugging and issuing system PM commands purposes, we also connect the board to a host Linux PC through a serial cable. We disable the DVFS feature on the board and fix the CPU frequency to be 1GHz.

BBB uses a device tree to describe its hardware, some devices on the board are disabled in the device tree. Based on our implementation in §5, we enable all the devices that can be supported by the 5 drivers we modify. Note that not all those devices are being used on BBB when the system is running, idle ones are supposed to be disabled at runtime if the runtime PM feature is enabled for them. We use the same device tree blob for all our experiments to make sure that all the measurements are on the same device hardware.

6.2 Effective power management

Table 6.1 shows the overall power consumption on the BBB using both our generic and the driver-provided `suspend/resume` functions. With generic context manager providing generic `suspend/resume` functions, we can achieve the same level of power savings as the original driver-based `suspend/resume` functions when doing both runtime PM and system PM.

For each result in Table 6.1, we measure the power consumption for 60 seconds

PM functions	runtime PM disabled	runtime PM enabled	suspend-to-RAM
Generic	904.80 \pm 2.72	757.05 \pm 2.55	243.45 \pm 0.50
Original	904.19 \pm 3.25	758.80 \pm 1.91	244.24 \pm 0.24

Table 6.1 : Overall power consumption on BBB (mW). The generic context manager can achieve the same level of power savings as the original driver-based PM. When measuring the power for runtime PM, we disable/enable the runtime PM feature only for the devices supported by our modified drivers.

with 5000 data samples per second. We then evenly divide all data samples into 10 pieces and calculate the average power consumption of each piece. The final results shown in the table are the means and standard deviations of the 10 average power consumption measurements. The standard deviations in the table are very small, but the power consumption within each piece does have a lot of variation when the system is running. We care about the power consumption in a long run, thus these variation are amortized by the average power consumption within each piece.

Runtime PM: When measuring the power consumption for runtime PM, we disable/enable the runtime PM feature only for the devices we list in §5 to test the runtime PM effectiveness on them. Runtime PM feature can be disabled and enabled by issuing *on* and *auto* commands to the *power/control* sysfs entry respectively for each device. Both the our generic and the original runtime **suspend/resume** functions can save ~ 146 mW when runtime PM feature is enabled. 97% of this power saving comes from the fact that the LCD controller is disabled at runtime. Note we only

test the effectiveness of the suspend/resume functions, we do not control when to disable/enable a device at runtime, this is done by the original device drivers.

System PM: When measuring the power consumption for system PM, we put the system into the *suspend-to-RAM* mode by issuing the command:

```
echo mem > /sys/power/state
```

When the system is suspended to this mode, on-chip devices in the wakeup domain will remain always on to wait for any wakeup events to resume the system. We resume the system by pressing a key on the keyboard from the Linux host machine, which will send a wakeup event to UART0 in the wakeup domain (shown in Figure 6.1). Both the generic `suspend/resume` functions and the original ones can put the system into the suspend-to-RAM mode that consumes ~ 243 mW. This shows that the generic `suspend/resume` functions are as effective as the original ones.

However, this 243 mW power consumption is very high for a mobile system in the suspend-to-RAM mode. The reason for this high power consumption is that all power rails (Figure 6.1) are left on by the original BBB kernel when the system is suspended. We enable the HDMI framer to convert the signals from the LCD controller, when the system is suspended this hardware is not turned off, which contributes ~ 90 mW power consumption. Moreover, VDD_3V3B and VDD_3V3AUX are used to supply power for on-board components such as LAN, eMMC and pin pads, they also consume a significant amount of power. After we disable the HDMI framer (as a result LCD controller is also disabled) and turn off the VDD_3V3AUX power rail, we reduce the

overall power consumption to 130 mW when the system is in suspend-to-RAM mode. But we are not able to turn off the VDD_3V3B rail because of the subtle PCB design on the board. We expect that even lower power consumption can be achieved if the VDD_3V3B is turned off.

As we discuss in §4, we can provide per-device knowledge to the generic context manager through the device instances to tune per-device PM behaviors. The original GPIO and I2C controller drivers don't provide system `suspend/resume` functions, therefore all the GPIO and I2C controllers will remain on when the system is suspended. But only GPIO0 and I2C0 in the wakeup power domain (PD_WKUP) are supposed to be always on. In our implementation, we tailor the device-specific knowledge for GPIO0 and I2C0 to avoid doing the system suspending and resuming, while letting other GPIO and I2C controllers use the generic system `suspend/resume` functions normally. In this way, we can disable more devices than the original driver-based PM. But the power savings for GPIO and I2C controllers are too trivial to be observed in our evaluation.

6.3 Reduced device driver complexity

We count the SLoC of `suspend/resume` functions before and after we move them out of the device drivers. As shown in Table 6.2, generic context manager reduces the driver complexity by completely removing or removing at least 40% of the `suspend/resume` code out of device drivers.

Device drivers	PM type	Original SLoC	Remaining SLoC	Reduction ratio
I2C	runtime	41	0	100%
	system	0	0	N/A
GPIO	runtime	175	59	66%
	system	0	0	N/A
MMC	runtime	193	104	46%
	system	76	45	40%
SPI	runtime	25	0	100%
	system	25	0	100%
LCDC	runtime	0	0	N/A
	system	35	13	63%

Table 6.2 : SLoC count for suspend/resume code that still remains in device drivers. Generic context manager can move most of the suspend/resume code out of the 5 drivers, as a result it reduces driver complexity.

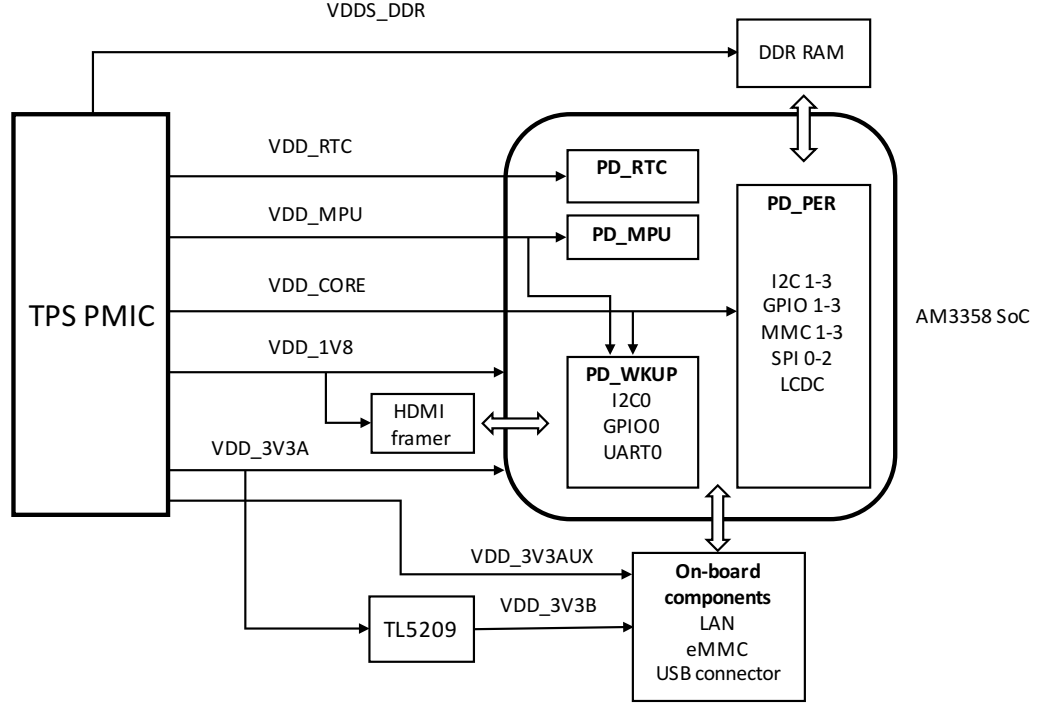


Figure 6.1 : Power rails on BBB, resources from [1,2]. Not all the devices in the power domains are showed in the figure. When the system is suspended, VDD_MPU and VDD_CORE will be scaled down to 0.95V, all other rails remain on. This causes the overall power consumption to be 243 mW when the system is suspended.

We use the same counting techniques as described in §3, i.e., we expand the function calls in `suspend/resume` functions to the scope of the driver file. Runtime `suspend/resume` functions could share code with system `suspend/resume` functions, in this evaluation we count them separately. In our implementation, we represent device-specific knowledge in C data structures. But we don't include the SLoC of those data structures in Table 6.2, assuming these data structures can be generated automatically from the device TRM.

As we explain in §4, not all `suspend/resume` code can be moved out of device drivers. Specifically in our evaluation, `suspend/resume` code for I2C and SPI con-

troller drivers is completely removed. Runtime suspend/resume code remained in the GPIO controller driver is for configuring remote wakeup. For the MMC controller driver, the code remained for runtime PM is a resume function that reinitializes the controller if there is a context loss when the controller is suspended; and the code remained for system PM is used to configure the voltage supply after resuming from the system PM. For the LCD controller driver, the remained code is for restoring the framebuffer content from the DRM framework.

The implementation of our generic context manager consists of 2338 lines of C code, among which 1053 lines are for generic suspend/resume implementation and register table definition. The remaining code are for device and driver instance registration and device accessing methods configuration, such as device register memory remapping and clock configuration, etc. Our generic context manager can support more drivers with only very minor modifications.

6.4 Latency

Our implementation of generic **suspend/resume** functions can incur additional latency when executing PM activities compared to the original driver-based ones. We use the ARM cycle counter registers to count the CPU cycles (at 1GHz) taken to execution both our generic and the original **suspend/resume** functions. Because we are focusing on moving the **suspend/resume** functions out from device drivers, we only measure the suspending/resuming execution time on the device driver side and the generic context

	runtime PM			system PM		
	I2C	GPIO	MMC	MMC	SPI	LCDC
original suspend	3646	2228	3702	2786	202	2093919
generic suspend	14796	22631	17363	5475	17285	2829146
original resume	4143	4469	3262	1373	7078	16574198
generic resume	11489	17927	17962	1823	2169	16745088

Table 6.3 : Execution latency of `suspend/resume` functions in CPU cycles, we configure the CPU frequency to be 1GHz, so the numbers are also in nanoseconds. Generic `suspend/resume` functions could be 1x to 9x slower than the original ones. But the absolute execution time is very small.

manager side. For comparison between the original driver-provided **suspend/resume** functions and our generic ones, we measure the execution time only if the original drivers provide the **suspend/resume** functions, as indicated by Table 5.1.

Our generic **suspend/resume** functions are 1x to 9x slower than the original ones. This is because the generic context manager implements a wrapper around normal **read/write()** functions to access various device registers in a generic way, which adds 50 to 100 cycles to each **read()** or **write()** call. Besides this, parsing the register tables mentioned in §4.4 to figure out the operations and values for each register access also adds a runtime overhead of 100 to 400 more cycles. The generic system **suspend()** for the SPI device in Table 6.3 is 85x slower than the original one, because it does much more than the original one. The original system **suspend()** for SPI is inadequate, it only selects a sleep state for SPI pins. Besides this, our generic system **suspend()** also saves necessary execution context and disengages the power and clock for SPI controllers by calling the APIs from the runtime PM framework. Also, because of this difference, the original system **resume()** takes a lot more time (more than 90% of the original system **resume()** execution time) than the generic one in the runtime PM framework to check the device’s runtime PM status.

Even though the ratio of the slowness of our generic **suspend/resume** functions is big, the actual time for **suspend/resume** functions execution is very small. Most of the **suspend/resume** functions execution time is under 20 μ s, except the LCD controller driver whose execution time is dominated by the framebuffer saving and restoring.

This time is negligible compared to the time period that a device is disabled or enabled, e.g., 50 to 100 ms [4]. The tolerance for the suspending/resuming latency depends on the Quality of Service (QoS) requirements from the device or system users. We defer the exploration of better data structures to represent device-specific knowledge that can reduce the `suspend/resume` functions execution latency to future work.

6.5 Limitations

As our implementation of the generic context manager is a prototype to show the feasibility of moving `suspend/resume` functions out of device drivers, it currently faces several limitations.

Our current generic context manager does not completely move all the suspend/resume code out of device drivers. Driver developers still need to spend considerable effort handling device-specific PM activities discussed in §4.3. Also, our proposed design only works for devices whose PM context is accessible to the CPU. Some devices such as GPUs and DSPs run their own software, usually the CPU does not have full access to their internal registers or memories. To power manage these devices, they should provide interface to expose their context to CPU or rely on their own software to manage their PM context. In the latter case, the CPU only issues commands to inform the software that the devices should be power managed.

Actually, with the presence of more and more ASICs, we argue that a standard

power management interface should be introduced to hardware design to simplify their power management code. With the standard hardware interface, device-specific power management happening in hardware is hidden from the device drivers, which can further advance moving power management code out of drivers and greatly simplify driver development.

Chapter 7

Related Work

Device drivers are notorious for their development difficulty, which is partially reflected by their relative portion in the kernel in terms of source lines of code, and by the prevalence and severity of their bugs. About 60% of mainline Linux kernel code is drivers [12]; more than 60% of kernel bugs identified in two studies across 10 years [11, 12] are from drivers. Driver bugs are famous for crashing the entire system [13] and subjecting it to security exploits [17, 36].

Our work is the first publicly known work that aims at moving the **suspend/resume** functions out of drivers. It also represents an important step toward simplifying device drivers and improving their reliability, following the strategy of moving things out of drivers.

7.1 Moving things out of driver

Over the past decade, the Linux kernel has moved many driver functions out of drivers and implemented them as device-independent frameworks, such as the common clock framework [24] and pinctrl framework [23]. Power management, nevertheless, has remained in the driver, despite the recent runtime PM framework that provides a reference counting mechanism for drivers to track their device usage. The authors

of [4] showed that many device drivers failed to use the Linux runtime PM framework properly. In particular, they failed to properly invoke the APIs that track if a device has pending tasks. They presented a tool to insert the API calls automatically based on trace-based analysis and an OS module that can infer if a device has pending tasks without driver’s help. Both solutions still rely on the driver to provide the `suspend/resume` functions. Our work complements this work toward device-independent power management by to move much of `suspend/resume` functions out of drivers.

7.2 Representing device knowledge

Many prior works have explored how to simplify driver development by redesigning the interfaces that drivers use to interact with both hardware and software.

The *device access interface* describes how the device can be accessed in software running on the CPU. Because a device’s programming interface consists of registers, the device access interface is about these registers. Driver code that provides the device access, or *access code*, is considered boilerplate code [37], tedious and error-prone [20]. Devil [19] and HAIL [20] sought to specify this interface in order to automatically generate the access code in C.

A device driver also exports services to the rest of the operating system and may use services provided by the rest of the OS. This *OS-driver* interface is about the interaction between the driver and the OS. Tingu [38] specifies the OS-driver

interface as a state machine that has ports to receive/send messages. As successor to Tingu, the Termite specification language [21,22] specifies both the device and OS interfaces as state machines.

How we represent device-specific knowledge is very similar to the existing approaches about device access interface. We can potentially exploit their techniques to describe device-specific knowledge using high-level domain specific languages (DSL) and automatically compile them into data structures. However, we do not aim at synthesizing the access code from the knowledge, rather, we make device access code generic.

Moreover, Barrelfish OS utilized a constraint logic programming language (CLP) to separate device configuration logic from the configuration mechanism [39]. This CLP-based declarative language offers more flexibilities to adapt new driver designs when both device configuration algorithms and device hardware are changing rapidly. Similar to their work, our generic PM logic can be considered as the configuration mechanism for power management, and the device-specific knowledge is the configuration logic. The separation between the device-specific knowledge and PM logic also helps to simplify PM code adaption in device drivers.

7.3 Move drivers out of kernel

An orthogonal strategy to deal with buggy drivers is to move them out of the kernel. LeVasseur *et al.* [40] run an unmodified driver with its own OS in a virtual machine

but to serve another OS. Doing so allows drivers developed for one OS to be used by another; importantly it isolates the second OS from the driver failure because the driver is running outside the second OS. SUD [41] emulates a Linux kernel environment in the user space to run unmodified device drivers in that emulated environment. It leverages the fact the device registers are memory mapped to provide direct device access to the driver. It also leverages the IOMMU and transaction filtering in PCI express bridges to control the memory operations from the device as directed by the driver.

Instead of moving a driver entirely out of kernel, one can also move part of a driver to the user space to improve system’s reliability and reduce its attack surface size. Microdriver [16] split a device driver into a high-performance *k-driver* and a low-performance *u-driver*. *k-driver* runs in kernel level at full speed to handle data path, while *u-driver* runs in user level at reduced speed to handle control path. Library driver [17] separated resource management and resource isolation in a driver, pushing resource management to an untrusted user-space library while keeping resource isolation running in the trusted OS kernel.

Related to moving things out of kernel, Nooks [13] isolates OS extensions such as drivers insight lightweight kernel protection domains which restrict extensions’ accesses to kernel memory and protect operating system from their faults.

7.4 Driver fault tolerance and correctness

Another line of previous work related to our goal of simplifying drivers is to improve their fault tolerance and enhance their correctness. Nooks [13] isolates drivers within kernel protection domains, and when it detects a driver is faulty, it recovers the driver by reloading and restarting it. *Shadow drivers* [14] employs record and replay to rollback failed drivers transparently from driver clients. FGFT [15] employs a finer-grained checkpointing mechanism than shadow drivers to rollback failed drivers. Importantly, it relies on the driver-supplied `suspend/resume` functions to checkpoint and restore device states. Dingo [42] leveraged the Tingu specification [38] of the OS-driver interaction to generate a runtime protocol observer that detects possible driver protocol violations. Ryzhyk *et al.* unified hardware verification with driver development to improve driver reliability [43]. One extreme approach to ensure driver correctness is to synthesize them from device knowledge automatically, as attempted by Termite [21] and Termite-2 [22].

Chapter 8

Concluding Remarks

By separating knowledge from logic for PM context management, we are able to make much of the PM context management device-independent and realize it out of device drivers. This leads to several directions that we should further explore. First, device-specific knowledge data structures should be generated automatically by device vendors. We can explore more standardized data structures to represent PM knowledge that covers as many devices as possible. The device-specific knowledge can be represented by a high-level descriptive language and compiled into the data structures by sophisticated compiler techniques. In our implementation, we manually study the original `suspend/resume` functions and separate the PM knowledge from the logic. As tons of different device drivers already exist in commodity operating systems, manual separation of PM knowledge from logic will take huge efforts. Thus, we can explore automatic techniques to extract PM knowledge from existing drivers, e.g., by monitoring their interactions with devices.

Second, running the generic context manager as a centralized module makes it possible to move the module onto low-power processors. When the system is put into sleep, runtime PM can be done by the low-power processors without waking up the power-hungry CPUs.

Furthermore, moving PM `suspend/resume` functions out of device drivers is part of our work to simplify device drivers. The Linux kernel has evolved such that more and more common frameworks are built to handle common resources such as power, clock, pinctrl, and so on. Following this trend, our generic context manager offers a way to redesign device driver software stack to make drivers simple, i.e., by folding as much knowledge as possible into data structures and building generic logic for the knowledge. Device drivers should only provide minimal interfaces for the device users to tap to exploit specific device functionalities. Other than that, device drivers only contain device-specific knowledge in data structures that are used to drive device-independent logic. Besides power management, there are other functions that can be moved out of device drivers such as device initialization.

Bibliography

- [1] *AM335x Sitara TM Processors Technical Reference Manual*. Texas Instruments.
- [2] G. Coley, *Beaglebone Black System Reference Manual*. Texas Instruments.
- [3] R. J. Wysocki, “Device power management.” <https://www.kernel.org/doc/Documentation/power/devices.txt>.
- [4] C. Xu, X. Lin, Y. Wang, and L. Zhong, “Automated OS-level device runtime power management,” in *Proc. ACM ASPLOS*, 2015.
- [5] “Android WakeLock Mechanism.” <https://developer.android.com/reference/android/os/PowerManager.WakeLock.html>.
- [6] K. Kim and H. Cha, “Wakescope: runtime wakelock anomaly management scheme for android platform,” in *Proceedings of the Eleventh ACM International Conference on Embedded Software*, p. 27, IEEE Press, 2013.
- [7] E. S. Raymond, *The Art of Unix programming*. Addison-Wesley Professional, 2003.
- [8] H. Esmailzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Computer Architecture*

(ISCA), *2011 38th Annual International Symposium on*, pp. 365–376, IEEE, 2011.

- [9] “I/o Kit Overview.” <https://developer.apple.com/library/content/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/Introduction/Introduction.html>.
- [10] “Windows Driver Model (WDM).” [https://msdn.microsoft.com/en-us/library/windows/hardware/Ff565698\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/Ff565698(v=vs.85).aspx).
- [11] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An empirical study of operating systems errors,” in *Proc. ACM SOSP*, 2001.
- [12] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, “Faults in linux: Ten years later,” in *Proc. ACM ASPLOS*, 2011.
- [13] M. M. Swift, B. N. Bershad, and H. M. Levy, “Improving the reliability of commodity operating systems,” in *Proc. ACM SOSP*, 2003.
- [14] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, “Recovering device drivers,” *ACM Trans. Comput. Syst.*, vol. 24, Nov. 2006.
- [15] A. Kadav, M. J. Renzelmann, and M. M. Swift, “Fine-grained fault tolerance using device checkpoints,” in *Proc. ACM ASPLOS*, 2013.
- [16] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha,

- “The design and implementation of microdrivers,” *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 168–178, 2008.
- [17] A. A. Sani, L. Zhong, and D. S. Wallach, “Glider: A gpu library driver for improved system security,” *arXiv preprint arXiv:1411.3777*, 2014.
 - [18] M. J. Renzelmann and M. M. Swift, “Decaf: Moving device drivers to a modern language,” in *USENIX Annual Technical Conference*, 2009.
 - [19] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller, “Devil: An IDL for hardware programming,” in *Proc. USENIX OSDI*, 2000.
 - [20] J. Sun, W. Yuan, M. Kallahalla, and N. Islam, “HAIL: a language for easy and correct device access,” in *Proc. ACM EMSOFT*, pp. 1–9, 2005.
 - [21] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser, “Automatic device driver synthesis with Termite,” in *Proc. ACM SOSP*, pp. 73–86, 2009.
 - [22] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij, “User-guided device driver synthesis,” in *Proc. USENIX OSDI*, 2014.
 - [23] “Linux pinctrl (Pin Control) Subsystem.” <https://www.kernel.org/doc/Documentation/pinctrl.txt>.
 - [24] “Linux Common Clock Framework.” <https://www.kernel.org/doc/Documentation/clk.txt>.

- [25] “Linux Runtime Power Management for I/O Devices.” https://www.kernel.org/doc/Documentation/power/runtime_pm.txt.
- [26] devicetree.org, “Devicetree specification release 0.1,” 2016.
- [27] A. Danial, “<https://github.com/aldanial/cloc>.”
- [28] <https://review.cyanogenmod.org/#/c/139054>.
- [29] M. Kerrisk, *The Linux programming interface*. No Starch Press, 2010.
- [30] “<https://www.kernel.org/doc/documentation/driver-model/design-patterns.txt>.”
- [31] T. Instruments, *OMAP543x Multimedia Device Silicon Revision 2.0*. 2015.
- [32] https://github.com/beagleboard/linux/blob/4.1/drivers/mmc/host/omap_hsmmc.c#L767.
- [33] https://github.com/beagleboard/linux/blob/4.1/drivers/tty/serial/8250/8250_omap.c#L1420.
- [34] <https://github.com/beagleboard/linux/blob/4.1/drivers/gpio/gpio-omap.c#L1319>.
- [35] M. S. Inc., “<https://www.msoon.com/>.”
- [36] “Linux Kernel i915 Driver Memory Corruption Vulnerability.” <https://tools.cisco.com/security/center/viewAlert.x?alertId=16920>.

- [37] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi, “Solving the starting problem: device drivers as self-describing artifacts,” in *Proc. The European Conf. Computer Systems (EuroSys)*, 2006.
- [38] L. Ryzhyk, I. Kuz, and G. Heiser, “Formalising device driver interfaces,” in *Proceedings of the 4th workshop on Programming languages and operating systems*, p. 10, ACM, 2007.
- [39] A. Schüpbach, A. Baumann, T. Roscoe, and S. Peter, “A declarative language approach to device configuration,” *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, p. 5, 2012.
- [40] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, “Unmodified device driver reuse and improved system dependability via virtual machines,” in *Proc. USENIX OSDI*, 2004.
- [41] S. Boyd-Wickizer and N. Zeldovich, “Tolerating malicious device drivers in linux,” in *USENIX Annual Technical Conference*, 2010.
- [42] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser, “Dingo: Taming device drivers,” in *Proc. The European Conf. Computer Systems (EuroSys)*, 2009.
- [43] L. Ryzhyk, J. Keys, B. Mirla, A. Raghunath, M. Vij, and G. Heiser, “Improved device driver reliability through hardware verification reuse,” in *Proc. ACM ASPLOS*, 2011.