Experiences with Power Management Enabling on the Intel Medfield Phone

R. Muralidhar, H. Seshadri, V. Bhimarao, V. Rudramuni, I. Mansoor,
S. Thomas, B. K. Veera, Y. Singh, S. Ramachandra
Intel Corporation

Abstract

Medfield is Intel's first smartphone SOC platform built on a 32 nm process and the platform implements several key innovations in hardware and software to accomplish aggressive power management. It has multiple logical and physical power partitions that enable software/firmware to selectively control power to functional components, and to the entire platform as well, with very low latencies.

This paper describes the architecture, implementation and key experiences from enabling power management on the Intel Medfield phone platform. We describe how the standard Linux and Android power management architectures integrate with the capabilities provided by the platform to provide aggressive power management capabilities. We also present some of the key learning from our power management experiences that we believe will be useful to other Linux/Android-based platforms.

1 Introduction

Medfield is Intel's first smartphone SOC built on a 32 nm process. The platform implements several key innovations in hardware and software to accomplish aggressive power management. It has multiple logical and physical power partitions that enable software/firmware to selectively control power to functional components and to the entire platform as well, with very low latencies.

Android OS (Gingerbread/Ice Cream Sandwich) supports Suspend-to-RAM (a.k.a S3) state by building upon the traditional Linux power management infrastructure and uses concepts of wake locks (application hints about platform resource usage) to achieve S3. The power management infrastructure in Android requires that applications and services request CPU resources with *wake locks* through the Android application framework and native Linux libraries. If there are no active wake locks, Android will suspend the system to S3.

While the S3 implementation in Android helps reduce overall platform power when the device is not actively in use, S3 state does not satisfy applications that require always connected behavior (Instant messengers, VoIP, etc., need to send "keep alive" messages to maintain their active sessions). Entering S3 will result in freezing these applications and connections timing out so the sessions will have to be re-established on resume. The Medfield platform allows such applications to be active and yet achieve good power numbers through S0ix, or Connected Standby states. The main idea behind S0ix is that during an idle window, the platform is in the lowest power state as much as possible. In this state, all platform components are transitioned to an appropriate lower power state (CPU in Cx state, Memory in Self Refresh, components clock or power gated, etc.). As soon a timer or wake event occurs, the platform moves into an "Active state", only the components that are needed are turned on, keeping everything else in low power state. S0ix states are completely transparent to user space applications.

Figure 1 illustrates how S0ix states impact platform power states and how this compares with traditional ACPI-based power management.

This paper is organized as follows. Section 1 is this introduction. Section 2 presents a background of the Linux and Android Power Management architecture. Section 3 describes the key Intel specific power management components on Medfield platform to achieve S3 and S0ix. In Section 3, we will describe our experiences with enabling overall Power management, challenges/issues with handling wake interrupts, handling suspend/resume/runtime PM in different device drivers, and some optimizations that we had to implement on

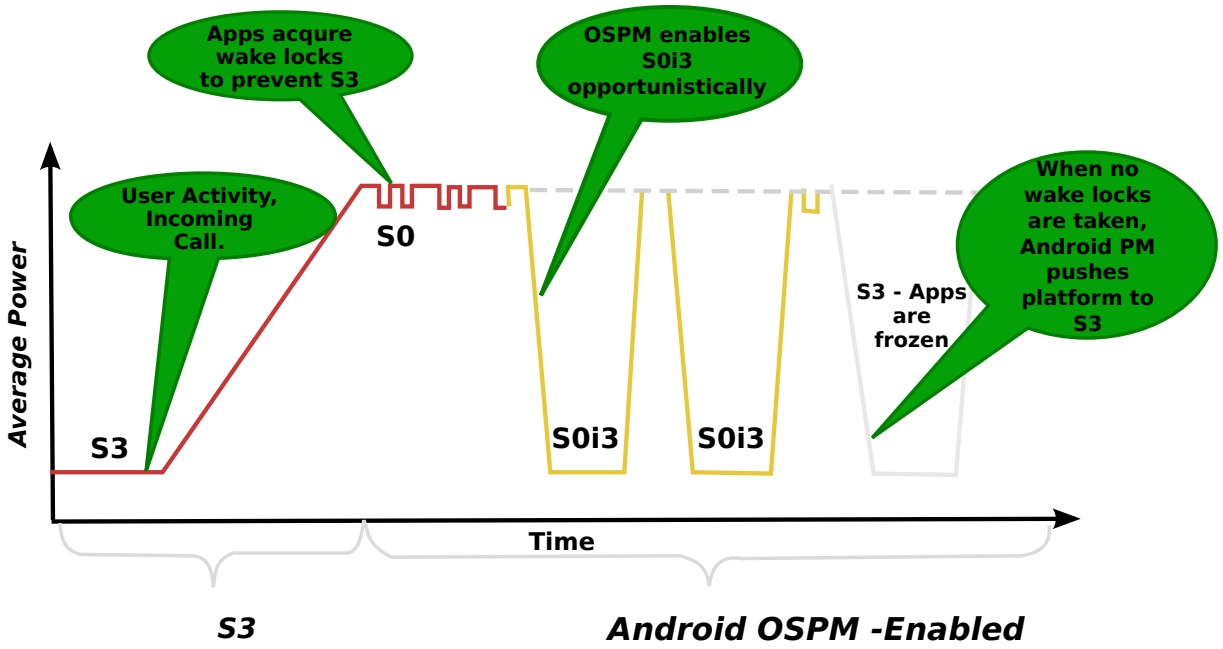

Figure 1: Platform Power States with S0ix and S3

the platform. We believe that some of these learning will be applicable to other Linux/Android-based SOC platforms as well.

2 Medfield Platform Power Management Architecture

Medfield (or Atom Z2460) is Intel's first 32 nm smartphone SOC; the Atom Saltwell core runs at up to 1.6 GHz with 512KB of L2 cache, a PowerVR SGX 540 GPU at 400 MHz, a dual channel LPDDR2 memory interface (PoP LPDDR2 (2 x 32 bit support)), and ISP from Silicon Hive, additional I/O, and an external Power Management delivery unit. Figure 2 shows the high level architecture of the Medfield SOC platform.

The Saltwell CPU is a dual-issue, in-order architecture with Hyper Threading support. The integer pipeline is sixteen stages long—the longer pipeline was introduced to help reduce power consumption by lengthening some of the decode stages and increasing cache latency to avoid burning through the core's power budget. There are no dedicated integer multiply or divide units, they are all shared with the floating point hardware. The CPU supports several different operating frequencies and power modes. At the lowest power level is its C6 state. Here the core and L2 cache are both power gated with their state saved in a lower power on-die SRAM. Total power consumption in C6 of the processor island is effectively zero. In addition to the 512KB L2 cache there is a separate 256KB SRAM which is lower power and on its own voltage plane. When Saltwell goes into its deepest sleep state, the CPU state and some L2 cache data gets parked here, allowing the CPU voltage to be lowered even more than the SRAM voltage. As expected, with hyperthreading the OS sees two logical cores to execute tasks on.

2.1 Core Linux Changes for x86 Smartphones

The Medfield platform is not PC-compatible in several aspects—no BIOS, no ACPI, no legacy devices, no PCI enumeration in South complex, no real IOAPIC, etc. Most of these changes are available in the upstream kernel now. Refer to [2], which discusses more details of the core kernel changes done for x86-based Intel platforms. This section briefly summarizes the key changes.

The following key changes were made to the underlying kernel in order to minimize changes from existing IA operating systems, and also provide software programming compatibility for key components of the platform (such as, PCI enumeration, IOAPIC interrupt controller, etc.):

1. Simple Firmware Interface (SFI) to replace ACPI

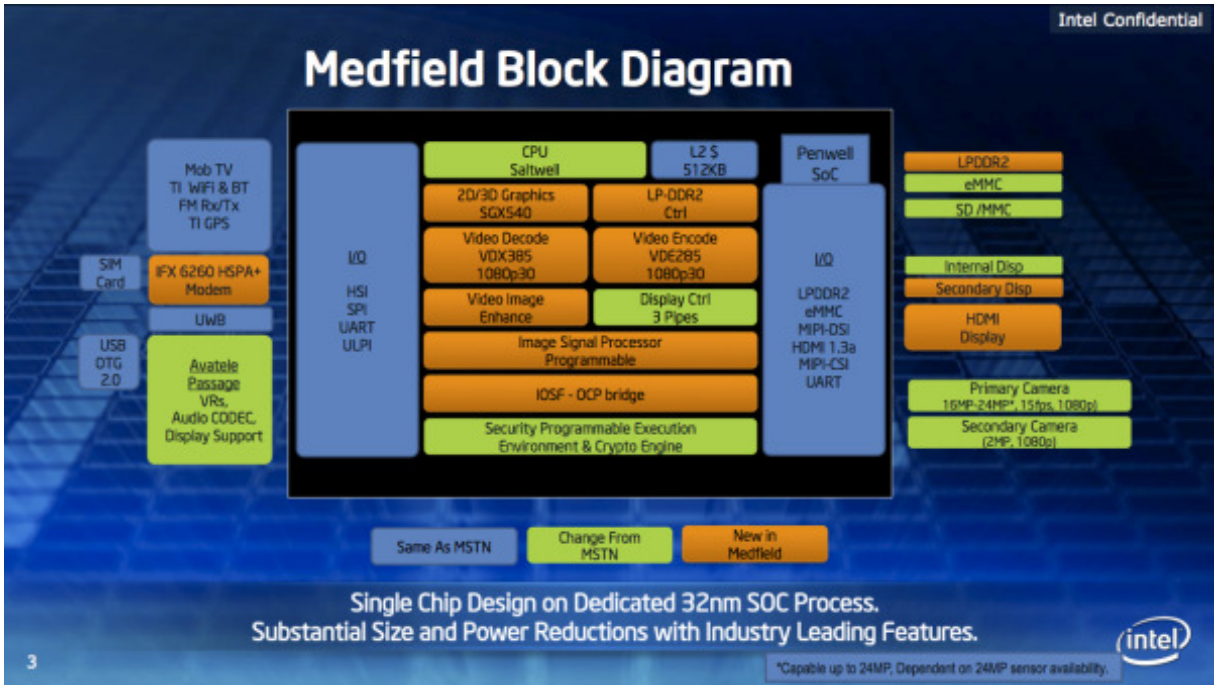

Figure 2: Medfield Platform Architecture

in order to report standard capabilities like CPU P-states, GPIOs, etc.

2. PCI Enumeration for South complex devices
3. IOAPIC Emulation

2.1.1 Simple Firmware Interface

Simple Firmware Interface (SFI) is a method for platform firmware to export static tables to the operating system. Platform firmware prepares SFI tables upon system initialization for the benefit of the OS (CPU P-states, GPIOs, for example). The OS consults the SFI tables to augment platform knowledge that it gets from native hardware interfaces, such as CPUID and PCI. More details on SFI can be found in [3].

2.1.2 PCI Enumeration

Penwell north complex devices are true PCI devices (Graphics, Display, Video encode/decode, ISP), but the South complex devices are fake PCI devices. All these south complex devices are enumerated as PCI devices through a PCI shim (Fake PCI MMCFG space written by firmware into main memory during platform boot) in the kernel. The PCI config space contains both true and fake PCI devices. MMCFG location is stored in SFI. Although this mechanism leverages existing device enumeration mechanism and reuses generic PCI drivers, this approach has its shortcomings in that it cannot detect device presence automatically. Also, PCI shim is read only, therefore, cannot handle writes to PCI config space.

2.1.3 Interrupt Routing and IOAPIC Emulation

Platform specific interrupt routing information is obtained from system firmware via PCI MMCFG space and SFI tables. Also, the south complex System controller Unit (SCU) maintains IOAPIC redirection tables that establish mapping between IRQ line and interrupt vectors.

3 Background: Linux and Android Power Management

Traditional ACPI-defined low power states for the platform are Hibernate to disk (S4) and Suspend to Ram (S3). A detailed treatment of the Linux power management architecture can be found in [5]. The kernel includes platform drivers responsible for carrying out

low-level suspend and resume operations required by particular platforms. The platform drivers are used by the PM Core, which is itself generic; it runs on a variety of platforms for which appropriate platform drivers are available, including ACPI-compatible personal computers (PCs) and ARM platforms. Additionally, Linux kernel 2.6.33 and beyond mandate that all device drivers implement Linux Runtime Power Management, which is a framework through which device drivers can implement autonomous power management when idle. This is aggressively used in Medfield platform. On Medfield Android, we only support S3 and subsequent references to standby mean only S3.

3.1 Linux Suspend Resume Flow

When the system goes into the S3 state, the phases are: `prepare`, `suspend`, `suspend_noirq`. This is illustrated in Figure 3 and described in detail in the Linux kernel documentation.

- `prepare` - This may prepare the device or driver in some way for the upcoming system power transition (for example, by allocating additional memory required for this purpose) but it should not put the device into a low-power state.
- The `suspend` methods should quiesce the device, save the device registers and put it into the appropriate low-power state. It may enable wakeup events.
- The `suspend_noirq` phase occurs after IRQ handlers have been disabled, which means that the driver's interrupt handler will not be called while the callback method is running. This method should save the values of the device's registers that weren't saved previously and will finally put the device into the appropriate low-power state. Most device drivers need not implement this callback. However, bus types allowing devices to share interrupt vectors, like PCI, generally need it.

When resuming from standby or memory sleep, the phases are: `resume_noirq`, `resume`, `complete`.

- The `resume_noirq` callback methods should perform actions needed before the driver's interrupt handler is invoked.
- The `resume` method should bring the device back to its operating state so that it can perform normal I/O.
- The `complete` method should undo the actions of the prepare phase.

3.2 Android Power Management Architecture

Android Power Management infrastructure is split across the User space and Kernel layer. *Wake Locks* form a critical part of the framework. A *Wake Lock* can be defined as a request by the applications and services for some of the platform resources (CPU, display, etc.). The Android Framework exposes power management to services and applications through the PowerManager class. All calls from applications to acquire/release wake locks into Power Management should go through the Android runtime PowerManager API.

Kernel drivers can register with the Android Power Manager driver so that they are notified immediately prior to power down or after power up—drivers must register `early_suspend()` and `late_resume()` handlers, which are called when display power state changes. Please refer to [1] and [4] for more details.

4 Power Management Architecture in Medfield

Medfield platform provides fine-tuned knobs for platform level power management and expects the Operating System Power Manager (OSPM) to direct most of these power transitions of the subsystem. OS Power managers like ACPI, APM, etc. traditionally directs the platform to various power states (S3/S4, for example) depending on different power policy set by the user. In Medfield, the OS Power Manager guides the power states that the subsystems and CPU need depending on the Power policy set by the user. The HW then makes the policy decision. This is done by dedicated Power Management Units (PMU) that reside on the Platform. This gives the flexibility of making finer power state transitions which are normally not possible through traditional OS power management methods.

4.0.1 Power Management Capabilities

As is well known, the major sources of power dissipation in CMOS devices, as described in [8] and [9] are:

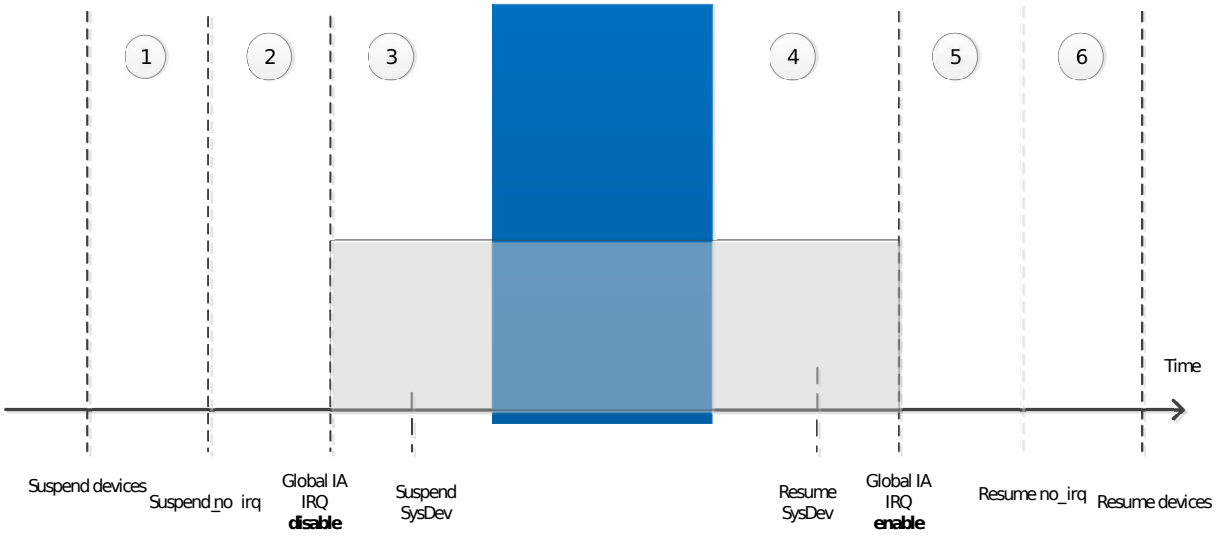

Figure 3: Linux Suspend Resume Flow

Switching power or *dynamic power* and *Leakage power*.

Switching or dynamic power represents the power required to charge and discharge circuit nodes. Broadly speaking, dynamic power depends on supply voltage (actually the square of supply voltage, $V^2 f$), clock frequency (f), node capacitance C (which in turn, depends on wire lengths), and switching activity factor (how frequently wires transition from 0 to 1, or from 1 to 0). Techniques such as clock gating are used to save energy by reducing activity factors during a hardware units idle periods. The clock frequency f, in addition to influencing power dissipation, also influences the supply voltage. Typically, higher clock frequencies will mean maintaining a higher supply voltage. Thus, the combined $V^2 f$ portion of the dynamic power equation has a cubic impact on power dissipation. Strategies such as dynamic voltage and frequency scaling (DVFS) try to exploit this relationship to reduce *(V, f)* accordingly.

Leakage power results due to current dissipation even when devices are not switching. The main reason behind this leakage is that transistors do not have ideal switching characteristics, and thereby leak a non-zero amount of current even for voltages lower than the threshold voltage. Hence power gating the entire logic (if possible) can ideally reduce the leakage power; this comes with additional responsibilities of saving/restoring the state, firewalling, etc.

The power management architecture in Medfield is built around these ideas aggressively that we can turn off subsystems without affecting the end user functionality and usability of the system. This is enabled by several platform hardware and software changes:

1. On die *clock and power gating* of subsystems
2. *Subsystem active idle states* that are OS transparent as well as driver managed
3. *Platform idle states* - extending idleness to the entire platform when all devices are idle

Device Power Management Capabilities - D0ix

All components, including the CPU, can be clock or power gated (individually, or as a combination). The CPU itself has its usual power states; C0 implies full power, full performance, and C6 is a deep sleep state where power is shut off to the entire CPU and state is saved in a small amount of active SRAM. The different power states supported by the Saltwell CPU are shown in Table 1.

Traditionally (according to ACPI, for example), subsystems/devices can be in active power state (D0) or in low power state (D1/D2/D3). Most subsystems/platforms implement D0 and D3, however, not many platforms/systems implement really active idle states, where the platform is active, but subsystems, even though are idle are in lower power state. In Medfield, devices can be in one of the following power states, traditionally called D-states:

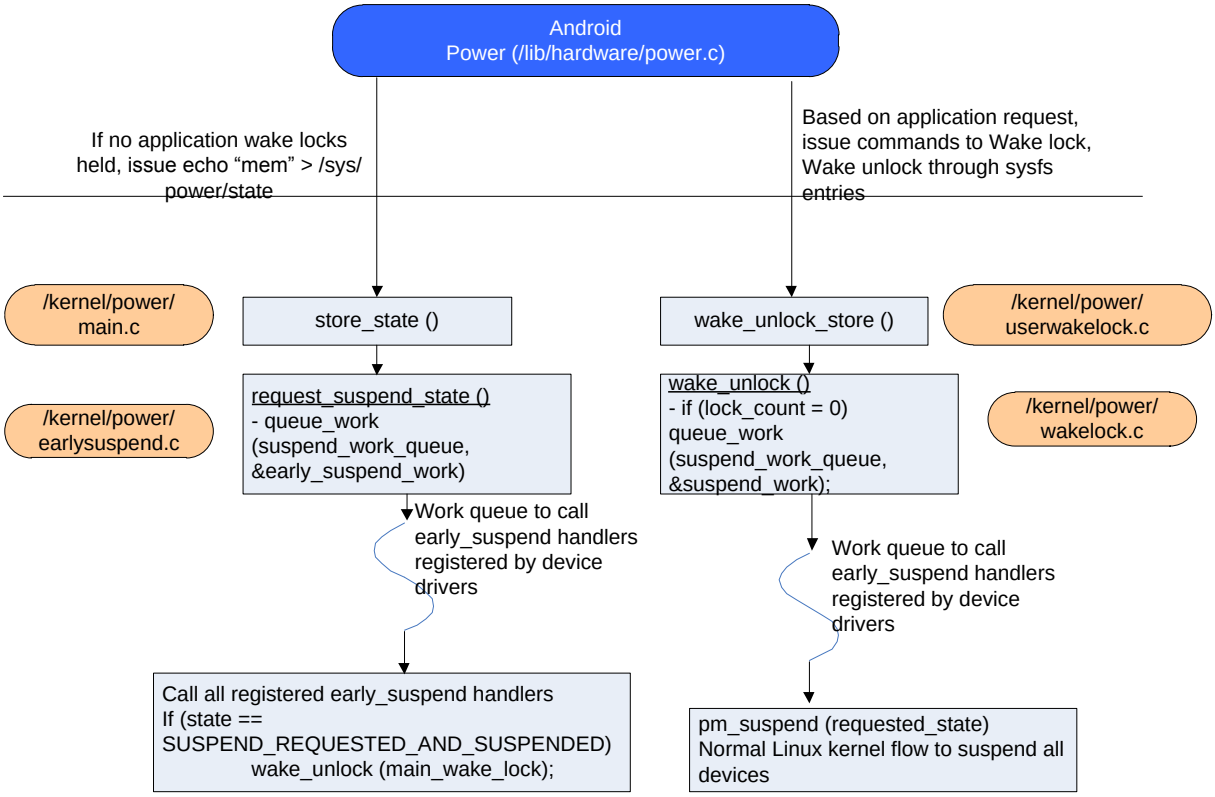

Figure 4: Android Suspend Resume Flow

Feature	C0 HFM	C0 LFM	C1-C2	C4	C6
Core Voltage	ON	ON	ON	ON	OFF
Core Clock	ON	ON	OFF	OFF	OFF
L1 Cache	ON	ON	Flushed	Flushed	OFF
L2 Cache	ON	ON	ON	Partial Flushed	OFF
Wakeup time	Active	Active	Least	More	Highest

Table 1: Summary of Saltwell CPU Power States

1. D0 - Normal operational state

2. D0i1 - OS-transparent clock gated state

3. D0i3 - Driver directed management of the subsystem with no OS control of the subsystem. The device driver coordinates and manages the subsystem state (and saves/restores state as needed) for power transitions.

4. D3 - OS directed management of the subsystem. The device driver is involved in the management of the subsystem and it must perform state retention and restoration in the driver. OSPM will manage transitioning of power state of the device and the device driver must be involved in the power state transition.

All devices will be managed through the runtime Linux power management infrastructure. Device drivers must implement D0i3 (driver managed autonomous power management) through the Linux Runtime power management framework, and aggressively (and intelligently) manage the power of their corresponding subsystems. Additionally, device drivers must also support standard Linux suspend/resume callbacks for implementing D3.

4.1 Power Management Architecture

The key components of power management architecture on Medfield are:

1. Standard cpuidle- and cpufreq-based CPU power and performance management components (native drivers and governors).
2. Platform-specific S0ix extensions to the cpuidle driver (intel_idle-based) for Medfield's Saltwell CPU
3. Power Manager Unit (PMU) driver - This driver interfaces with both North and South Complex Power Management Units (PMUs). It also provides platform-specific implementation of deep idle states to the intel_idle-based processor drive and coordinates with the rest of the platform using standard kernel Power Management interfaces like PM_QOS, Linux Runtime PM, etc.
4. PMU Firmware that coordinates power management between the Platform PMUs: P-UNIT for north complex (CPU, Gfx blocks, ISP), and SCU for south complex (everything else: IO devices, storage, comms, etc.)

CPUIDLE driver performs idle state power management. It plugs into existing CPU Idle infrastructure and extends current intel_idle processor driver for the Medfield CPU (code-named Saltwell). It also exposes new platform idle states deeper than traditional C6—these actually correspond to deep idle states for the entire platform, when there is sufficient idleness on the platform. More details about cpuidle can be found in [6].

CPU frequency is managed by the cpufreq driver. The Medfield cpufreq-based P-state driver uses the existing cpufreq infrastructure and exposes the CPU frequency states to the governors. The most common/generic cpufreq governor is the ondemand governor. ondemand is a dynamic in-kernel cpufreq governor that can change CPU frequency depending on CPU utilization. It was first introduced in the linux-2.6.9 kernel. It has a simplistic policy that provides significant benefits to the platform by making use of fast frequency-switching features of the processors to effectively manage their frequencies depending on the CPU load. For a good overview of how DVFS support is provided by these generic Linux modules, please refer to [7].

The PMU driver communicates with the CPU idle driver, platform device drivers, and the PMU firmware to coordinate platform power state transitions. Based on the guidance/hint from idle prediction, the PMU driver opportunistically extends CPU idleness to rest of the platform. In order to do this most efficiently, all device drivers must also be implementing and autonomous power management through Linux Runtime power management. The PMU driver provides a platform-specific callback to the CPU idle framework so that long periods of idleness can be extended to the entire platform. Once CPU and devices are all idle, this driver programs the north and south complex PMUs to implement the required power transitions. The state we enter is called a **S0ix** state.

Android S3 states are directly mapped to S0i3, the only difference being that timers are disabled in S3 state (as compared to S0ix where OS timers can wake up the platform). This is illustrated in Table 2.

PMU driver performs the following actions to emulate S3 over S0i3 :

1. *Program Wake configuration*: PMU driver disables timers as wake source, therefore only events like USB or Comms events will cause a platform wake
2. *Prepare for S3*: Here PMU driver triggers CPU state to be offloaded to a separate SRAM and issues a command to the SCU to enter S0i3.
3. *Enter C6* on both CPU threads with MWAIT instruction. This will guide the CPU to a package-level C6, thereby allowing the PMUs to proceed with S0ix entry sequence in firmware.

With the above actions, the platform enters S3 (S0i3 with timers disabled). It is to be noted that the CPU state that was saved includes everything until the point of MWAIT instruction execution so that on resume, the CPU will start executing from the next instruction after MWAIT. In some cases entering package-level C6 might still fail (break interrupt for example). In that case, SCU will wait for a timeout period before aborting the S0ix/S3 entry.

On exit from S3 the PMU driver gets an interrupt with status register specifying the wake source. This is followed by the actual device wake interrupt. During the resume flow, PMU driver thread will resume devices and trigger thaw_processes() and resume all the devices.

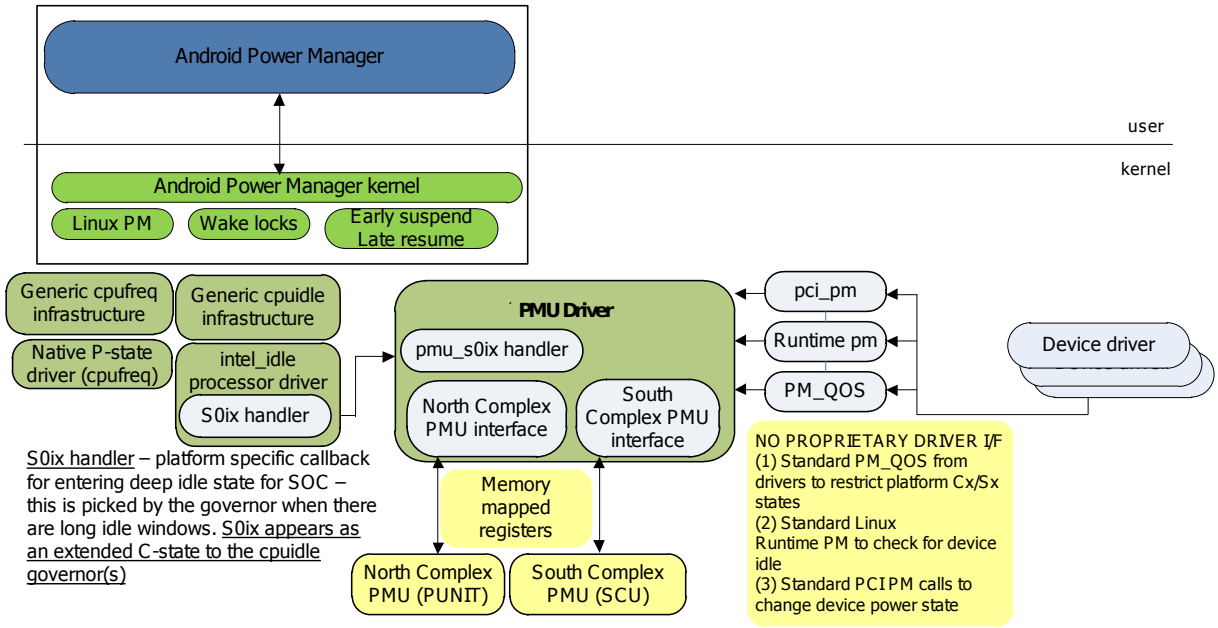

Figure 5: Medfield Platform Architecture

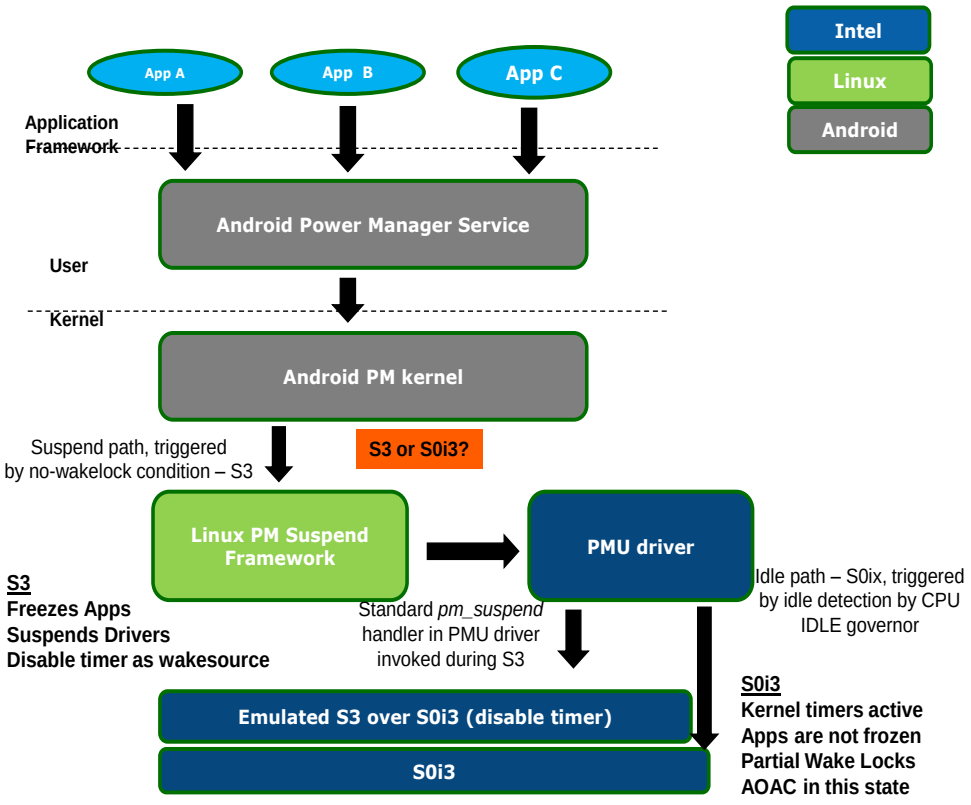

Figure 6: Implementing Android Power Management in Medfield

Islands	S0:C0-C6	S0i1	S0i3	S3
CPU	C-state dependent	OFF	OFF	OFF
C6 SRAM, Wake logic	ON	ON	OFF	OFF
DDR	ON/Self-refresh (SR)	SR	SR	SR
Power Manager	ON	ON	OFF	OFF
Graphics	ON/power gated (PG)	PG	OFF	OFF
Video Decode	ON/power gated (PG)	PG	OFF	OFF
Video Encode	ON/power gated (PG)	PG	OFF	OFF
Display Controller	ON/power gated (PG)	PG	OFF	OFF
Display	ON	OFF	OFF	OFF
Device drivers	ON/D0ix	D0i3	D0i3	D3
Applications	Active	Active	Active	Frozen

Table 2: What is on in S0, S0ix, S3?

5 Experiences with Enabling Power Management

This section summarizes some of the most important learning we had enabling power management on the Medfield platform. Some of this learning are relevant to other Linux/Android-based SOCs as well.

5.1 Runtime Power Management Implementation in Device Drivers

All Medfield device drivers implement the Linux Runtime PM framework, whereby drivers autonomously detect their idleness, and guide their corresponding devices to a low power state. Since runtime PM was relatively a new subject in the Linux, we had to spend a significant amount of time in making sure that we have the right implementation in all the drivers. This was one of the key elements for getting the standby functionality stable and robust.

1. **Idle Detection**: Due to the absence of a general rule to detect idleness, we had to establish a process to detect idleness specific to a driver/device. As an example, I2C driver implemented runtime PM based on a rule that if the I2C bus is not being used by devices for a certain idle time, it would put itself into a low power state. As soon as platform sensors were enabled (which were hanging off the I2C bus), none of the sensors allowed to enter a deep sleep state as they were constantly accessing the bus. We had to fine tune the idle detection value to something more optimal that would allow the platform to enter extended idle periods.
   Recommendation: *Idle detection would be more effective if done through a combination of hardware capability (OS-visible, device specific idle/activity counters in HW) and software (guidance from OS/drivers) that will allow tuning/optimizations.*

2. **Runtime PM callbacks**: All the PCI drivers had implemented legacy suspend and resume handlers and had also implemented runtime PM—these two cannot co-exist (if not implemented correctly)—this led to conflict between the device state as maintained by Runtime PM core and Standard Linux kernel PM Core (resulting in kernel panics during suspend/resume phase). This was subsequently fixed manually in all such offending device drivers.
   Recommendation: *All drivers must implement power management correctly as mandated by the Linux kernel recommendations, and must also take into account the new features being added to the kernel as the power management support therein evolves and matures.*

5.2 Interrupt handling

1. **Accessing hardware devices after resuming from D0ix/S0ix**: Device drivers must ensure that corresponding hardware is powered up before accessing device registers. For example, when an interrupt lands on the USB driver it would first try to check if the interrupt was for itself by accessing registers in the USB host controller. The device

driver must ensure that the hardware is powered up before accessing any registers. Specifically, device drivers were modified to move their hardware access code outside the IRQ handler into a kernel bottom half handler and ensuring that the hardware is powered up by doing a `pm_runtime_get_sync()` function.
Recommendation: *Wakes and interrupt delivery logic must be foolproof and the device drivers must also be intelligent to handle such cases.*

2. **Implementing Correct PM Functions**: Some devices can have multiple ways in which interrupts are triggered—in-band and out-of-band through a GPIO. One such device was HSI. We observed that HSI was causing hangs during entry path—when the driver's `suspend` function was invoked, the driver had suspended its device. But when a sideband wake occurred just a little later in the S3 entry phase, it had already suspended and lost its state, thereby causing a kernel panic. This was fixed by having the HSI driver implement `suspend_noirq()` handler which would ensure that no interrupts would land unexpectedly.
Recommendation: *Device drivers must follow the Linux kernel power management recommendations and implement all the relevant callbacks for suspend/resume and runtime PM.*

3. **Enabling Wake Interrupts**: We faced issues with wakes happening from power button, WLAN wakes, etc, from an OS perspective, the interrupt seemed to be lost when the driver had resumed from S3. What was happening was this: when platform resumes from S3, all resume handlers get called (in some sequence). If the default kernel IRQ handler does not see any registered IRQ handler for a specific interrupt (which would have happened during suspend phase where driver deregisters its IRQ handler), it handles it by default and sends an EOI down to the IOAPIC. Thus, such interrupts were handled by default by the kernel since the drivers had not resumed yet. The Linux kernel has support for such conditions—device drivers can indicate using the `IRQF_NO_SUSPEND` flag that its IRQ handler should not be completely removed in S3. If a driver sets this flag, the kernel will invoke the corresponding IRQ handler on high priority, even before the resume handlers are called.
Recommendation: *Drivers with wake capability must use `IRQF_NO_SUSPEND` flag and implement `suspend_noirq()` handlers if there can be multiple (in-band, out-of-band) wake sources.*

5.3 Optimizing for power and performance

A lot of work went into optimizing the platform for Power and Performance (PnP) for all the important use cases on the phone. This section summarizes some of the key experiences and learning.

1. Optimizing platform wakes: A bulk of optimizations around platform power and performance came from optimizing the number of wakes that bring the platform out of standby states. These optimizations spanned firmware, device driver, middleware and applications.

2. S0ix Latency Optimizations: Optimizing standby (S0ix as well as S3) latencies is a critical component of ensuring that the penalty for entering/exiting standby is amortized by the benefits of the low power achieved in those states.

3. Performance optimizations: A lot of optimizations were done across applications and middleware for fine tuning different aspects of performance. For example, we fine tuned the `ondemand` frequency governor, to get the most optimum thresholds for the platform. The threshold ratios (80/20, for example) correspond to how fast the platform can go to higher frequencies and the increments of coming down. We fine tuned the thresholds for the platform based on different characteristics. While we eventually achieved an optimal setting for the platform, clearly this seems to be in the domain of heuristics and is more empirical rather than really knowing what thresholds are best. Currently there are a lot of ongoing discussions to optimize and overhaul the `cpufreq` and `ondemand` governor infrastructure. For future platforms, we believe this is a good area to invest and optimize.

5.4 Tools

All of the above fixes and optimizations would not have been possible without proper hardware and software tools.

1. **Voltage rail level analysis**: A setup with a detailed data acquisition system (DAQ) to acquire rail level power consumption is a MUST when we are dealing with power optimization of handheld devices. Many of the above analysis and optimization was done with such a setup.

2. **Tools like `ftrace` and `powertop`**: Ftrace and Powertop are tools which are already in use within the Linux community. The former helps us with profiling the code which causes CPU activity and the latter helps us in analyzing the wake sources (both software and hardware).

   There were also internal tools that helped for debugging, analyzing device D0ix residencies, memory bandwidth/utilization, etc.

6 Summary

This paper described the architecture, implementation and key learning from enabling aggressive power management on Medfield. The paper explained the standard Linux and Android Power Management architecture and the key architectural enablers for aggressive power management on Medfield—standard Android PM (S3) as well as S0ix, Intel's innovation in HW/SW that enables aggressive low power idle states. Finally we presented some of the key learning from platform-wide power management enabling and optimizations that we believe are important to other SOCs.

7 Acknowledgments

Many teams in Intel have been instrumental in enabling Power Management on Medfield in various phases of architecture, design, pre- and post-silicon validation, software integration and optimization, etc. The authors would like to acknowledge the following individuals/teams (in no particular order): Bruce Fleming, Belli Kuttanna, Ajaya Durg, Ticky Thakkar, Randy Hall, Kalyan Muthukumar, Padma Apparao, Richard Quinzio, the entire Power management and power/performance team, Robert Karas, Jon Brauer, Sreedhara DS, Abhijit Kulkarni, Srividya Karumuri, Pramod HG, Rupal Parikh, Ryan Pinto, Mark Gross, Yanmin Zhang, Nicolas Roux, Pierre Tardy, Christophe Fiat and many others who have helped out in different phases/aspects of Power Management debug/enabling.

References

[1] Android Developer Reference, *http://developer.android.com*

[2] Jacob Pan, *Porting the Linux kernel to x86 MID Platforms*, Embedded Linux Conference 2010.

[3] Simple Firmware Interface, *http://www.simplefirmware.org*

[4] R. Wysocki, *Technical background of the Android Suspend Blockers Controversy*, http://www.lwn.net/images/pdf/suspend_blockers.pdf.

[5] L. Brown, R. Wysocki, *Suspend to RAM in Linux*, In Proceedings of the Ottawa Linux Symposium 2008.

[6] V. Pallipadi, A. Belay, S, *cpuidle: do nothing, efficiently*, In Proceedings of the Ottawa Linux Symposium 2007.

[7] V. Pallipadi, A. Starikovskiy, *The ondemand governor: past, present and future*, In Proceedings of Linux Symposium, 2006.

[8] A. Chandrakasan, S. Sheng, and R. Brodersen, *Low-power CMOS digital design, 1992*.

[9] A. Ghosh, S. Devadas, K. Keutzer, and J. White, *Estimation of average switching activity in combinational and sequential circuits. In Proceedings of the 29th ACM/IEEE conference on Design automation*, Pages 253âĂŞ259. IEEE Computer Society Press, 1992.

[10] Android PowerManagement, `http://developer.android.com/reference/android/os/PowerManager.html`