Linux Power Management Details

80-VR652-1 B

QUALCOMM Incorporated
5775 Morehouse Drive
San Diego, CA 92121-1714
U.S.A.

Revision History

Version	Date	Description
A	Jun 2009	Initial release
B	Feb 2010	Numerous changes were made to this document. It should be read in its entirety.

Note: The BSP information contained in these training materials and presented in connection with these training materials is to be used solely in connection with Qualcomm ASICs, related software and documentation.

Contents

Introduction

Introduction

- This document discusses, in detail, the Linux Power Management (PM) frameworks used.
- It also describes the responsibilities of device drivers for maximum power savings.
- This document focuses on how device drivers plug into the Linux PM frameworks in MSM7xxx/QSD chipsets.
- It shows the tight interaction between the modem and the apps for PM.
- This document does not discuss debugging of PM features (refer to [Q2]).

Introduction (cont.)

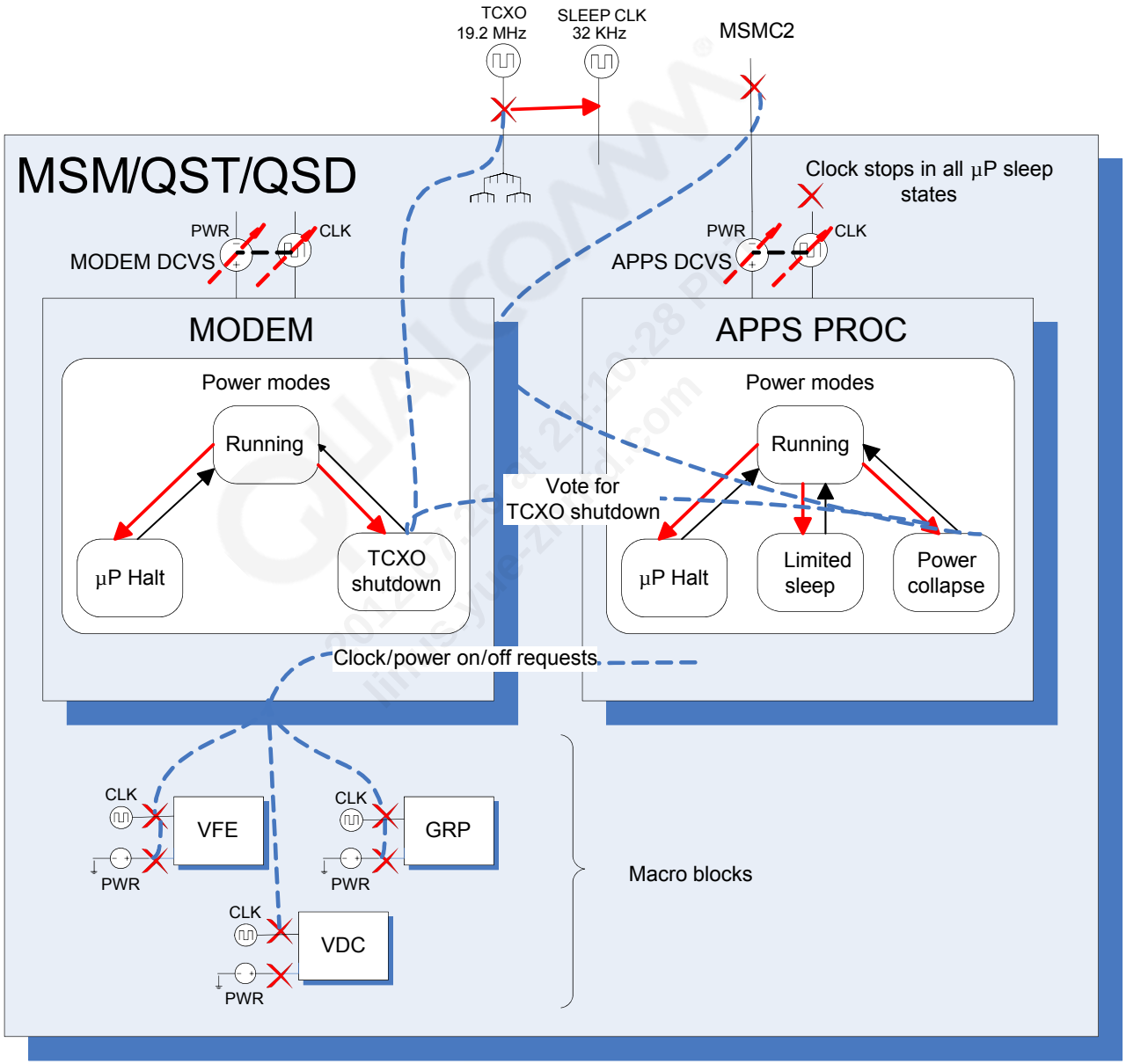
- Power Management is critical for mobile devices with small batteries (~1000 mAh), to extend the life of a battery charge.
- PDA functionalities that dominate mobile devices make it more challenging to keep average current consumption low, on top of modem functionality.
  - Music playback time may be as important as modem standby/talk time specification from carriers.
- The integrated application processor enables tight coupling of PM strategy with the modem processor to optimize overall current consumption.
- Tight coupling also complicates implementation.

Introduction (cont.)

- On Linux QSD/MSM7xxx targets, Qualcomm's proven, efficient, power-saving technologies are integrated with the existing Linux PM framework to achieve the best power savings in a very dynamic mobile environment.
- Qualcomm PM technologies integrated in Linux
  - Dynamic sleep mechanism
  - Dynamic Power modes and Sleep states
- Linux PM technologies
  - Linux tickless kernel
  - Linux core PM subsystem
  - Linux CPUFREQ subsystem
  - Linux Quality of Service (QoS) for PM (pm_qos_params)
  - Wakelock (introduced in Android™ kernel)
- Qualcomm modem PM features

Linux Power Modes

QSD/MSM7xxx PM Overview



Linux Power Modes – Overview

- Power modes
  - Running
  - Sleep
    - MSM Sleep = Apps power collapse + modem TCXO shutdown
    - Limited Sleep (Apps power collapse only) = Apps power collapse + Apps votes against modem TCXO shutdown
    - SWFI (only) = Apps executes SWFI instruction; no Apps power collapse or modem TCXO shutdown
    - Spins = Apps CPU spins; no Apps power collapse; no modem TCXO shutdown
  - Suspend
    - Apps power collapse + modem TCXO shutdown + off state for all hardware devices (clocks will be off, see next bullet)
    - Drivers notified of pending suspend and will go into lower Power mode and disable clocks

Linux Power Modes – Sleep

- Conditions for Sleep modes
  - Interrupt
    - Wake-up interrupt – Able to wake up apps from power collapse
    - Nonwake-up interrupt – Unable to wake up apps from power collapse
      - » This type of interrupt prevents AP from Idle power collapse
  - Latency
    - Time of entering and exiting Low Power mode
    - Driver's PM_QOS latency requirement (MIN) is checked against
      - » Biggest Latency (msm_pm_data[MSM_PM_SLEEP_MODE_POWER_COLLAPSE].latency in arch/arm/mach-msm/board-xxxx.c)
        – Apps power collapse + Apps votes for modem TCXO shutdown
      - » Intermediate Latency (msm_pm_data[MSM_PM_SLEEP_MODE_POWER_COLLAPSE_NO_XO_SHUTDOWN].latency)
        – Apps power collapse + Apps votes against modem TCXO shutdown
      - » Small Latency (msm_pm_data[MSM_PM_SLEEP_MODE_WAIT_FOR_INTERRUPT].latency)
        – Apps clock ramps down and SWFI
      - » [smaller than Small Latency – Spin]

Linux Power Modes – Sleep (cont.)

- Conditions for Sleep modes (cont.)
  - Residency
    - Time threshold (breakeven point) to save power next timer event is checked
  - Clocks are remotely monitored by clkrgm/Modem for TCXO shutdown
    - All clocks are disabled/enabled via proc_comm by clkgrm/Modem

Linux Power Modes – Sleep (cont.)

- MSM Sleep (Apps power collapse + apps votes for modem TCXO shutdown)
  - Entered from CPU idle upon these conditions
    - No nonwake-up interrupt enabled
    - No clock from TCXO source enabled
    - Next timer event > CONFIG_MSM_IDLE_SLEEP_MIN_TIME
    - Minimum interrupt latency > CONFIG_MSM_TCXO_SHUTDOWN_LATENCY
  - Apps Processor saves ARM® registers, Memory Management Unit (MMU) registers (used by virtual memory Linux), and warm-boot entry point
  - Apps Processor notifies Modem Processor
    - Requests power collapse + OK to TCXO shutdown
  - Modem turns off Apps Processor subsystem (Apps power rail turned off)
    - As a result, Apps Processor CPU, Interrupt Controller, timers, and caches are lost; RAM remains intact

Linux Power Modes – Sleep (cont.)

- MSM Sleep (cont.)
  - Modem Processor monitors Apps Processor interrupts and General-Purpose Input/Outputs (GPIOs) configured as wakeup interrupt sources
    - When wake-up interrupt fires (e.g., keypress), it handles interrupts, wakes up Apps Processor, and passes interrupts to Apps Processor
    - Wakes up for shared interrupt or GPIO interrupt firing, timer expiration, RPC call
  - Modem might decide to shut down TCXO
    - All modem subsystems and Apps Processor must vote TCXO off
    - Enough time to shut down TCXO prior to next timer expiration on both modem and Apps Processor
    - If so, modem configures MPM, the always-on hardware, to monitor interrupts and GPIOs that are wakeup sources

Linux Power Modes – Sleep (cont.)

- MSM Sleep (cont.)
  - Upon wakeup for Apps Processor
    - Modem Processor handles interrupt, restores TCXO, if needed, and powers up Apps Processor
    - Apps Processor starts warm boot and restores registers and MMU
    - Clocks, interrupt hardware, timer hardware restored on exit
- Limited Sleep – Apps power collapse only, no TCXO shutdown
  - Entered from CPU idle upon these conditions
    - No nonwake-up interrupt enabled
    - Next timer event > CONFIG_MSM_IDLE_SLEEP_MIN_TIME
    - Minimum interrupt latency > CONFIG_MSM_POWER_ COLLAPSE_LATENCY
  - Apps Processor saves ARM registers, MMU registers (used by virtual memory Linux), and warm-boot entry point

Linux Power Modes – Sleep (cont.)

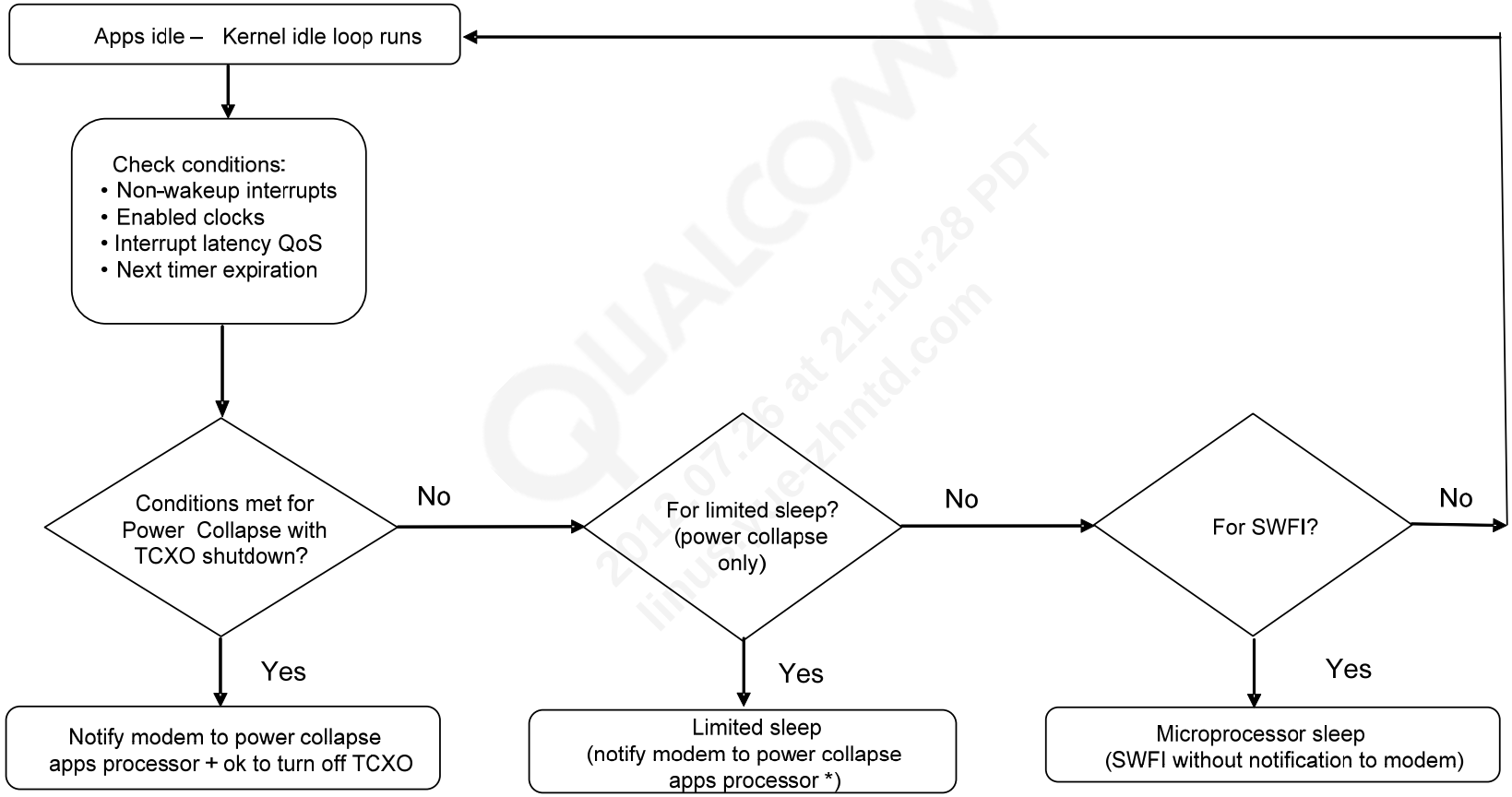
- Limited Sleep – Apps Power Collapse only, no TCXO shutdown (cont.)
  - Apps Processor notifies modem
  - Modem turns off Apps Processor subsystem
  - Apps power rail powered off – Apps Processor CPU, interrupt controller, timers, and caches are lost; RAM remains intact
  - Modem monitors Apps Processor interrupts and GPIOs
  - Modem monitors keypresses – When keypress interrupts fire, modem handles interrupts, wakes up Apps Processor, and passes interrupts to Apps Processor
  - Modem must wake up Apps Processor; wakes up for shared interrupt or GPIO interrupt firing, timer expiration, RPC call
  - Apps Processor starts warm boot and restores registers and MMU
  - Clocks, interrupt hardware, timer hardware restored on exit

Linux Power Modes – Sleep (cont.)

- Apps SWFI
  - Entered from CPU idle upon these conditions
    - Non-wakeup interrupt enabled
    - Next timer event <= CONFIG_MSM_IDLE_SLEEP_MIN_TIME
    - Minimum interrupt latency > CONFIG_MSM_SWFI_LATENCY
  - No modem intervention
  - Snapdragon™ Core CPU/Apps Processor halted
  - Any interrupt on Apps Processor causes exit

Sleep Decision Logic – Apps Processor

- Sleep decision algorithm in arch_idle()



Note: Power collapse can occur with or without voting for TCXO shutdown.

Linux Power Modes – Suspend

- Suspend
  - Entered from user application context by issuing suspend command through sysfs (/sys/power/state)
    - By key press from user (supported by Android UI)
    - After timeout or inactivity (supported by Android UI)
    - For details, refer to [Q2]
  - Drivers notified of suspend via suspend() callback
  - Drivers must enter their lowest Power mode
  - Apps CPU enters Power-Collapse state
  - Modem can enter TCXO shutdown
  - Since it is the longest sleep with power collapse and Low Power (off) mode for all hardware, this mode saves the most power

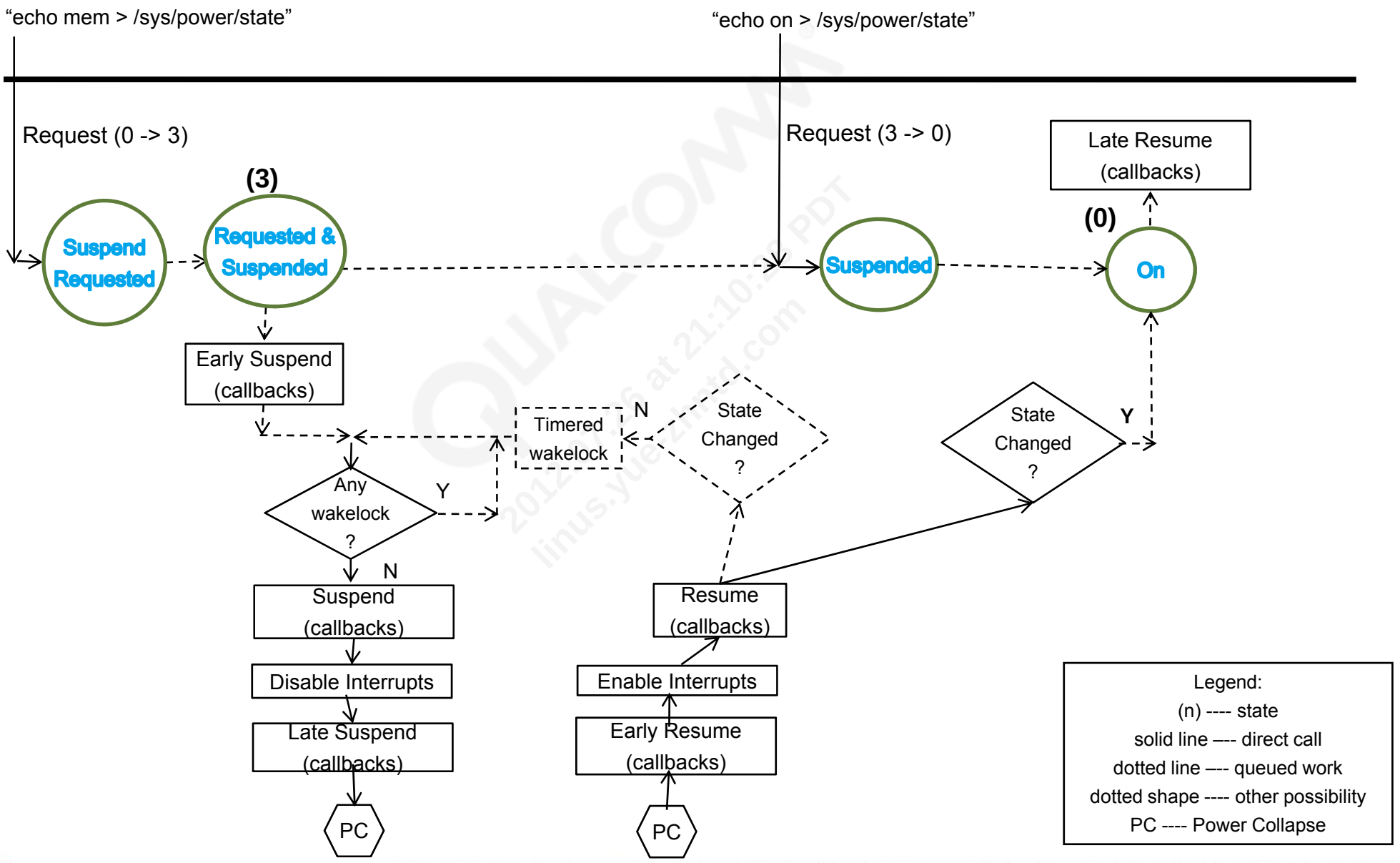
Early Suspend and Late Resume

Early Suspend and Late Resume

- Added in the kernel for Google/Android™
  - Early suspend – Before normal suspend (refer to figure on next slide)
    - Drivers notified of early suspend via callback
    - Drivers can put hardware into Low Power mode ahead of normal suspend callback
  - Late resume – After regular resume (refer to figure on next slide)
- Users (some drivers currently using early_suspend/late_resume)
  - Display driver
  - Keypad driver
  - Battery driver

Note: Regular suspend can be held off by wakelocks, but early_suspend is always called, even when there are wakelocks (more details in next few slides).

Early Suspend and Late Resume (cont.)



Early Suspend and Late Resume (cont.)

- Brief flow of the figure
  - "*echo mem > /sys/power/state*" causes suspend request
    - Makes the kernel platform call early_suspend () callbacks on all drivers registered for early_suspend()
    - A kernel thread is then spun, which will check for all wakelocks held; this thread uses a timer
      - » No wakelock → real suspend (suspend() callbacks on all drivers called)
    - Interrupts are disabled
    - late_suspend() callbacks called, then apps goes into power collapse

Early Suspend and Late Resume (cont.)

- Brief flow of the figure (cont.)
  - When resuming, early_resume() callbacks called
    - Interrupts may be enabled as needed
    - resume() callbacks are called on drivers
    - From here, only UI events (for example, "*echo on > /sys/power/state*") will cause late_resume() callbacks to be called
      - This causes late_resume() on the display driver, which eventually turns on LCD
    - In some cases, display should not be turned on and kernel should go back into suspend (e.g., RPC call wakes up the Apps and the call is handled by the kernel, and then kernel can go back into suspend; hence, no need to wakeup/resume the LCD)

Wakelock for Suspend

Wakelock for Suspend

- Added in the kernel for Google/Android
- To give drivers (kernelspace/userspace) a chance to finish the transaction that is in process, before going into suspend
- Users (some drivers currently using wakelocks)
  - USB
  - Keypad
  - MMC
- Suspend/wakelock contention
  - When suspend operation has already started while locking a wakelock, suspend operation will be aborted as long it has not already reached suspend_late stage
    - Locking a wakelock from an interrupt handler or a freezeable thread always works, but if wakelock is locked from a suspend_late handler, an error should be returned from that handler to abort suspend

Wakelock for Suspend (cont.)

- Driver APIs and their usage
  - Add a wakelock variable to driver state and call wake_lock_init.

```
struct state {
    struct wakelock wakelock;}
init() {
    wake_lock_init(&state->wakelock, WAKE_LOCK_SUSPEND,
       "wakelockname");
```

  - Before freeing the memory, wake_lock_destroy must be called

```
uninit() {
    wake_lock_destroy(&state->wakelock); }
```

  - When the driver determines that it needs to run (usually in an interrupt handler) it calls wake_lock

```
wake_lock(&state->wakelock);
```

  - When it no longer needs to run, it calls wake_unlock

```
wake_unlock(&state->wakelock);
```

Wakelock for Suspend (cont.)

- Driver APIs and their usage (cont.)
  - Calls wake_lock_timeout to release the wakelock after a delay

```
wake_lock_timeout(&state->wakelock, HZ);
```

    - This works whether the wakelock is already held or not. It is useful if the driver woke up other parts of the system that do not use wakelocks, but still need to run. This should be avoided whenever possible, since it will waste power if the timeout is long, or may fail to finish needed work if the timeout is short.

Wakelock for Suspend (cont.)

- User space APIs and their usage
  - Write "lockname" or "lockname timeout" to /sys/power/wake_lock lock and, if needed, create a wakelock (the timeout here is specified in nanoseconds)
  - Write "lockname" to /sys/power/wake_unlock to unlock a user wakelock
  - Do not use randomly generated wakelock names, as there is no API to free a user space wakelock

QoS Provided by PM

QoS Provided by PM

- Try to fill the gap between performance and power saving
  - New API on kernel version 2.6.25
- Algorithms
  - Minimum
  - Maximum
- QoS parameters (or services) (like nodes in NPA)
  - PM_QOS_CPU_DMA_LATENCY
    - Use MIN algorithm in sleep decision
  - PM_QOS_SYSTEM_BUS_FREQ (added by Qualcomm)
    - Use MAX algorithm in clock driver

QoS Provided by PM (cont.)

- APIs
  - Register
    - pm_qos_add_requirement(qos-parameter, driver_name, value/PM_QOS_DEFAULT_VALUE)
    - User space – qos_fd = open ("/dev/[qos-parameter]")
  - Request a QoS
    - pm_qos_update_requirement(qos-parameter, driver_name, new-value)
    - User space – write (qos_fd, new_value)
    - Call this before entering QoS-sensitive code section (many times/different values)
  - Remove all previously requested QoS
    - pm_qos_update_requirement (parameter, driver_name, PM_QOS_DEFAULT_VALUE)
    - User space – write (qos_fd, -1)
    - Call this after exiting QoS-sensitive code section
  - Unregister
    - pm_qos_remove_requirement(parameter, driver_name)
    - User space – close (qos_fd)

CPUFREQ

CPUFREQ – Linux Dynamic PM

- CPU frequency scaling (CPUFREQ) is PM method used in Running mode
- Switching of CPU frequency is governed by these governors in static or dynamic manner
  - Performance – CPU runs at static maximum frequency
  - Powersave – CPU runs at static minimum frequency
  - User space – Determined by static user space program
  - Ondemand – On-demand dynamic governor sets target frequency based on CPU busy/idle statistics
  - Conservative – Conservative dynamic governor sets target frequency, based on CPU busy/idle statistics
- Uses MSM-dependent CPU driver underneath
  - AXI speed (vote) is tied to CPUFREQ table using perf-level
  - SVS (and AVS on 8k) is tied to CPUFREQ table

CPUFREQ – Linux Dynamic PM (cont.)

- Sysfs interface – */sys/devices/system/cpu/cpu0/cpufreq/**
- Advantages
  - Users can change governors (refer to [Q2] for details)
  - Users can change speeds for user space governor
  - Users can change parameters for dynamic governors
    - Up threshold
    - Down threshold
    - Sampling rate
  - SVS and AVS can be tied with ACPU speed
- Disadvantages of dynamic governors
  - Reactive and timer based – Through a deferred timer
  - Default up/down thresholds (80/50) do not fit for all
  - Default sampling rate (200 ms) does not fit for all
  - Do not know how to scale BUS clock

CPUFREQ – Ondemand Governor

- Kernel times – user; nice; system; softirq; irq; idle; iowait; steal
- Busy time – user + system + irq + softirq + steal [+ nice]
- Load % – Busy time/total time
- Parameters
  - sampling_rate: [200 ms – between 100 ms and 100 sec]
  - up_threshold: [80 %]
  - down_threshold: [70 % – changed to 50 for MM]
  - powersave_bias: [1 – between 0 and 1000]
  - ignore_nice_load: [0] (1: nice = idle; 0: nice = busy)
- Algorithm
  - Up = MAX * [(1000 – powersave_bias) / 1000]
  - Down = Cur * (load% / down_threshold) * [(1000 – powersave_ bias) / 1000]

CPUFREQ – Conservative Governor

- Kernel times – Same as Ondemand
- Busy time – Same as Ondemand
- Load % – Same as Ondemand
- Parameters
  - sampling_rate: [500 ms – between 250 ms and 250 sec]
  - up_threshold: [80 %]
  - down_threshold: [20 %]
  - sampling_down_factor: [1 – between 1 and 10]
    - sampling_rate * sampling_down_factor -> down_sampling_rate
  - freq_step: [5 %]
  - ignore_nice_load: [0] (1: nice = idle; 0: nice = busy)
- Algorithm
  - Up = Cur + MAX * (freg_step / 100)
  - Down = Cur – MAX * (freq_step / 100)

Power-Saving Guidelines

Guidelines for Power Savings

- All drivers
  - Must support suspend
  - Do not set periodic timers
    - Processors woken up on next timer expiration, so this kills power savings
  - Take advantage of deferred timers to alleviate requirement to wake up for your timer
  - Use inactivity timer to turn off
    - Clocks
    - Nonwakeup interrupts
  - Actively manage
    - VREG / MPP / GPIO

Driver Responsibilities

- Driver responsibilities
  - Suspend
    - Register the device driver with Linux kernel and support .suspend and .resume callbacks
    - Put device in lowest Power mode in .suspend function
      - Device hardware and any VREGs/GPIOs/MPPs used by the device
    - Release wakelock
  - Idle power collapse
    - When not in use, driver should disable its interrupts, GPIOs, and clocks so that Apps Processor can enter power collapse and Modem Processor can enter TCXO shutdown
    - If possible, driver should use deferrable timer or range timer, instead of regular timer

Driver Responsibilities (cont.)

- Driver responsibilities (cont.)
  - CPUFREQ
    - Design and write power-optimized drivers; debugging and optimizing power on later stages is an ordeal
    - Avoid burst CPU requirements from drivers, if possible
    - When using ondemand governor, CPU frequency will jump the highest when load is above the up_threshold; use judgement when executing loops, relinquish CPU as much as possible; this spreads out the load and provides better power numbers
    - Test for power and performance as new features are added
    - Test drivers setting ondemand as the governor and compare it against user space at a reasonable speed; if ondemand seems to be running at a higher frequency than it should, then
      - Try tweaking the ondemand parameters (up_threshold, down_threshold, sampling_rate, scaling_min_freq, etc.)
      - Review loops and spin locks in your code

Driver Responsibilities (cont.)

- Driver Responsibilities (cont.)
  - How to handle resume from suspend
    - Register device driver with Linux kernel and support .resume callback
    - In general, resume code should undo everything that was done in suspend code
      - » If driver saves hardware settings and powers down hardware in suspend, it should power up hardware and restore hardware settings
      - » If driver disables services in suspend, it should re-enable the services
    - Driver should delay enabling of interrupts, clocks, GPIOs, vreg, etc., for as long as possible, until needed
      - » For example, if I^2C driver disables I^2C interrupt in suspend, driver can probably skip re-enabling of I^2C interrupt in resume; instead, I^2C driver will re-enable the interrupt when I^2C transaction is requested
      - » Good drivers should disable resources (interrupt, clocks, timers, etc.) when not needed and re-enable them when they are needed

Driver Responsibilities (cont.)

- Driver responsibilities (cont.)
  - Wakelocks
    - Recommended that drivers should not use wakelock, unless needed
      - When kernel goes into regular suspend, i.e., after all wakelocks have been released, kernel will call suspend (and suspend_late) function of each registered driver (refer to diagram for early suspend)
      - In suspend(), driver should put respective hardware into suspended state; if driver cannot do this, e.g., if it is waiting for an outstanding transaction to finish, it should hold a wakelock so that kernel will not enter regular suspend
      - Whether driver will need to use wakelock depends on whether the driver can finish (and finish quickly) all its work in the suspend function; if it can, it does not need wakelock; if it cannot, it will need wakelock

Driver Responsibilities (cont.)

- Driver responsibilities (cont.)
  - Early_suspend/late_resume
    - Drivers should try to use the early-suspend/late-resume mechanism
      - If a driver holds a wakelock, it is strongly recommended to register for early-suspend, so that it is notified via early-suspend callback when the suspend command is issued
      - Once driver receives early_suspend notification, it should stop accepting new transactions/requests, finish the outstanding one, and release the wakelock
      - If driver does not hold a wakelock, it means driver is OK with regular kernel suspend; most likely, driver should be able to perform its job in suspend, instead of early-suspend
      - There may be special cases where driver needs to go into Low Power mode before regular kernel suspend starts; in these cases, early-suspend is a solution

Device Low-Power Mode APIs

- Use the following APIs to allow sleep or avoid current leakage when
  - In driver suspend function
  - Not in use
- Clock APIs
  - Include <linux/clk.h>
  - Use clk_disable () when clock is not needed, so suspend or sleep (TCXO shutdown) can proceed; power saved when clocks turned off
- Voltage Regulator (VREG) APIs
  - Include <mach/vreg.h>
  - Use vreg_disable () to disable voltage regulator on PMIC
  - Use vreg_set_level () to set voltage regulator at lower voltage level

Device Low-Power Mode APIs (cont.)

- Multipurpose Pin (MPP) APIs
  - Include <mach/mpp.h>
  - Use mpp_config_digital_out() to set MPP pin on PMIC to correct state
    - Example
      - Output logic level – MSME (1.8 v), MSMP (2.6 v), MMC (3 v), VDD (3.6 v)
      - Output control – High, low (0 v), MPP input, inverted MPP input
- GPIO Top-Level mode MUX (TLMM) APIs
  - Include <mach/gpio.h>
  - Use gpio_tlmm_config() to set driver GPIOs to correct state
    - Configurations
      - Enable/disable
      - Input/output
      - Pull-up/pull-down/keeper/no pull
      - Drive strength (current) – 2 mA to 16 mA (example)

Apps and Modem Interaction

Apps/Modem Processor Interaction

- Close interaction between both processors during Apps Processor power-saving operations
- Modem controls TCXO shutdown procedure
  - Based on votes from Apps Processor and other modem subsystem voters
  - Modem uses Modem Power Manager (MPM) to disable/enable TCXO and to change voltage configuration
- Modem reroutes and detects interrupts, normally handled by Apps Processor, during TCXO shutdown and apps power collapse
- Modem shuts down Apps Processor power, based on Apps Processor decision
- Modem restores Apps Processor power, based on timer expiration or interrupts
- Modem programs Apps Processor DVS table in hardware for security reasons

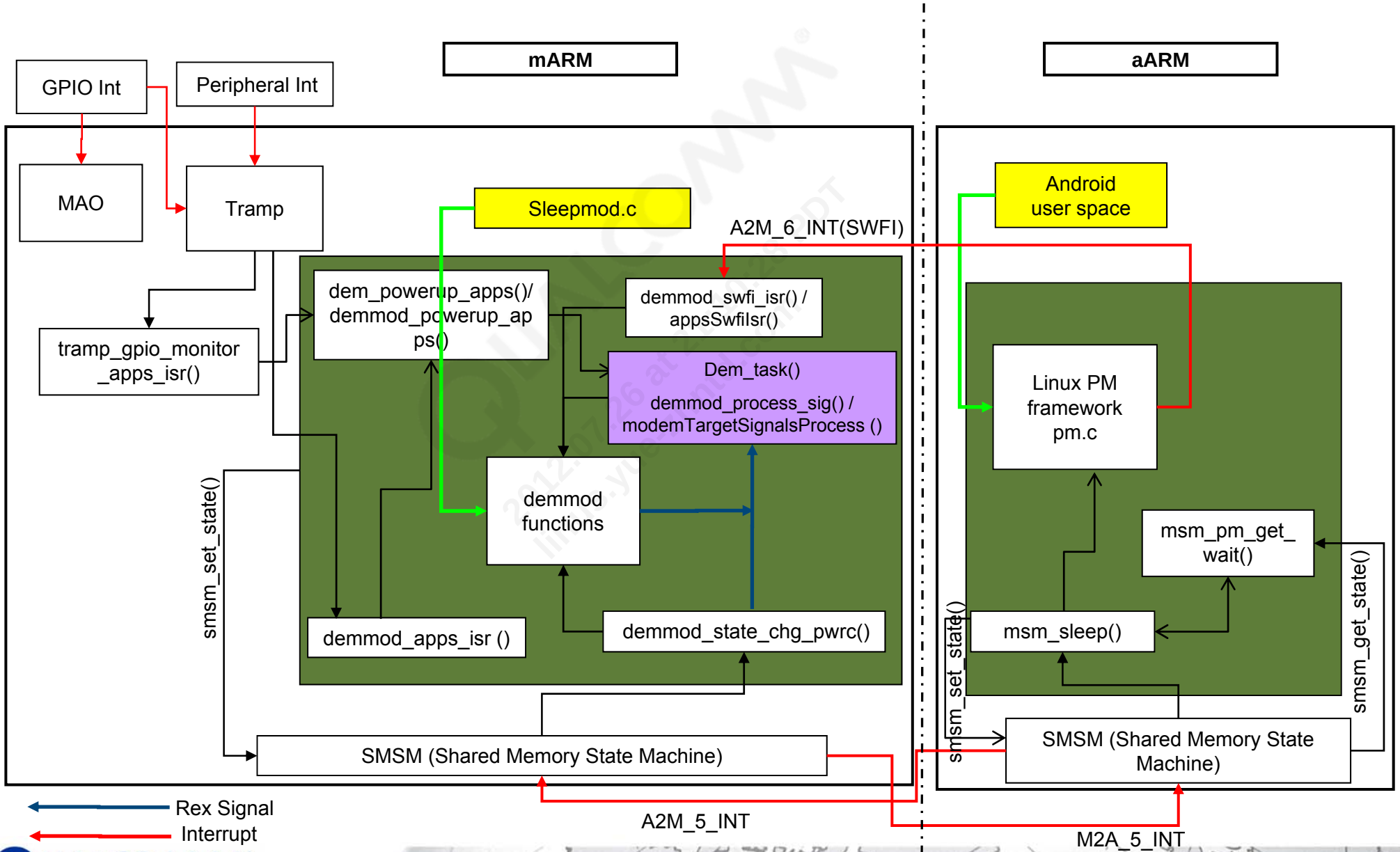
Apps/Modem Processor Interaction (cont.)

- Shared Memory (SMEM) used by Apps Processor (Snapdragon Core CPU/Apps Processor) and modem to exchange state information during SWFI and power-collapse procedures
  - Change of state (Modem « Apps Processor)
  - Interrupts configuration for rerouting
  - Apps Processor sleep/power-collapse duration
- Interrupt
  - Modem and Apps Processor notify each other of state change during power-collapse procedure via IRQ interrupt and polling
  - During apps power collapse, all interrupts on Apps Processor are disabled, except for M2A6 interrupt, which is used by Modem Processor to abort Apps Processor power collapse

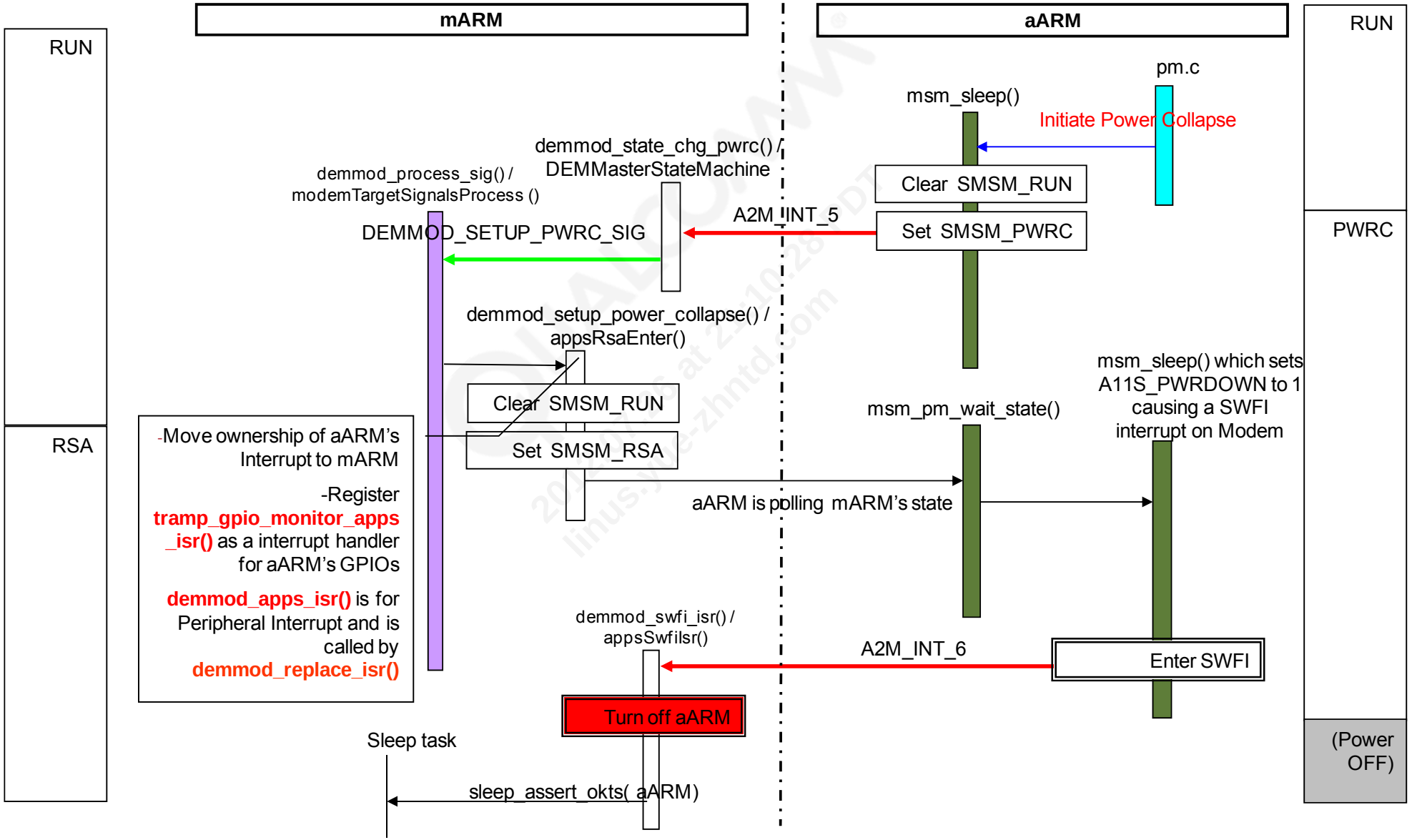
Apps/Modem Processor Interaction (cont.)

- Apps Processor reset vector (physical address of 0x0)
  - Modem wakes up Apps Processor after true power collapse
  - Apps reset vector at address 0x0 mapped by Linux after cold boot
  - Jump instruction placed at 0x0 before going to power collapse
  - Jump instruction jumps to Linux warm-boot code

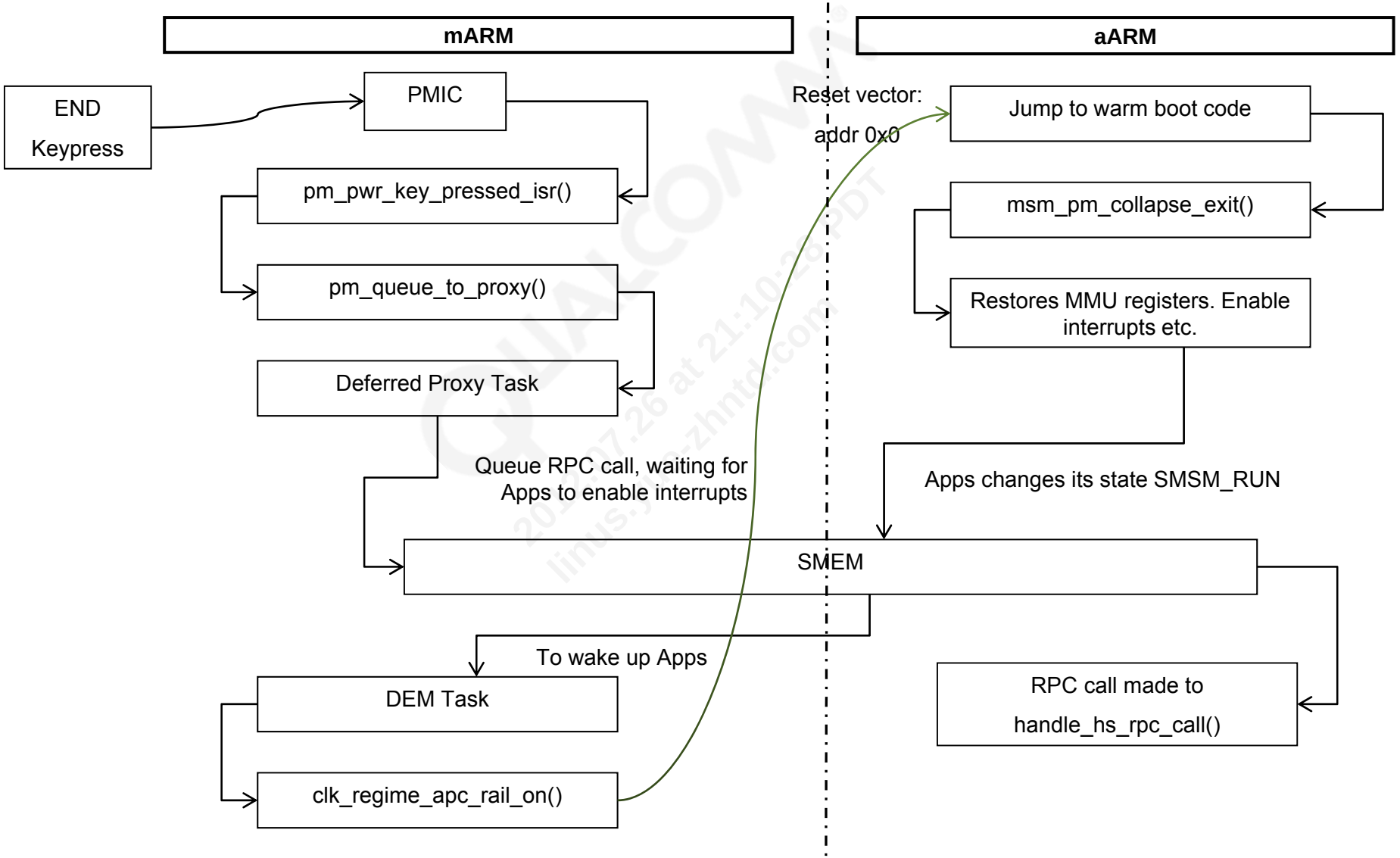
Detailed Modem and Apps Interaction



Enter Power Collapse (Suspend/Idle)

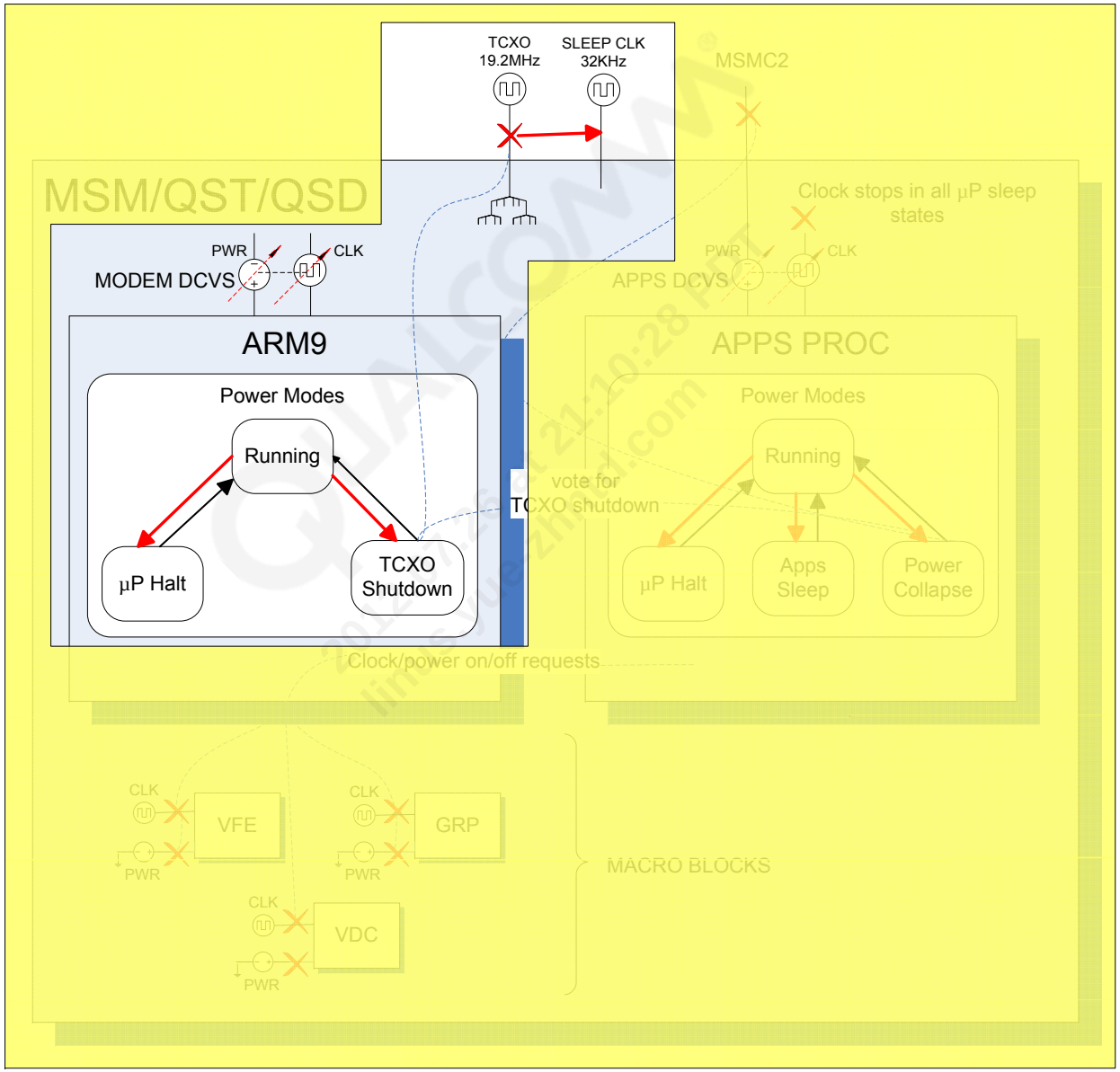


END Keypress to Wake up



Modem Power Management

Modem Processor PM Overview



Modem PM Features

- Modem – General power savings
- Modem – Sleep mechanism
- Modem – Sleep decision
- Modem – MPM
- Modem – TCXO shutdown
- Modem – Enter TCXO shutdown procedure
- Modem – Exit TCXO shutdown procedure
- Modem – Hard macro power collapse

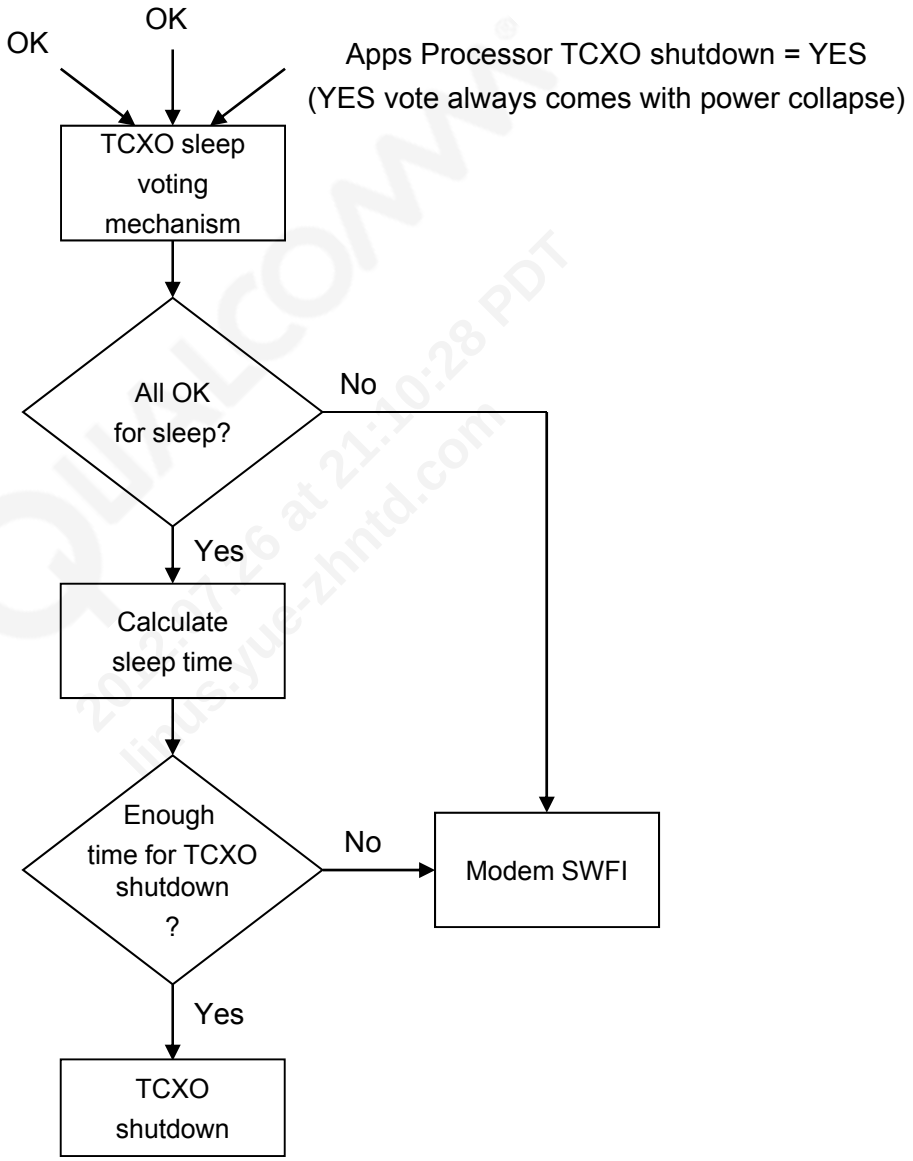
Modem – General Power Savings

- SWFI
- TCXO shutdown via MPM, with voltage reduction during sleep
- Hard macro power collapse – Video Front-End (VFE), Graphics Engine (GRP), and Video Codec (VDC)
- Foot switches to minimize leakage of DSPs

Modem – Sleep Mechanism

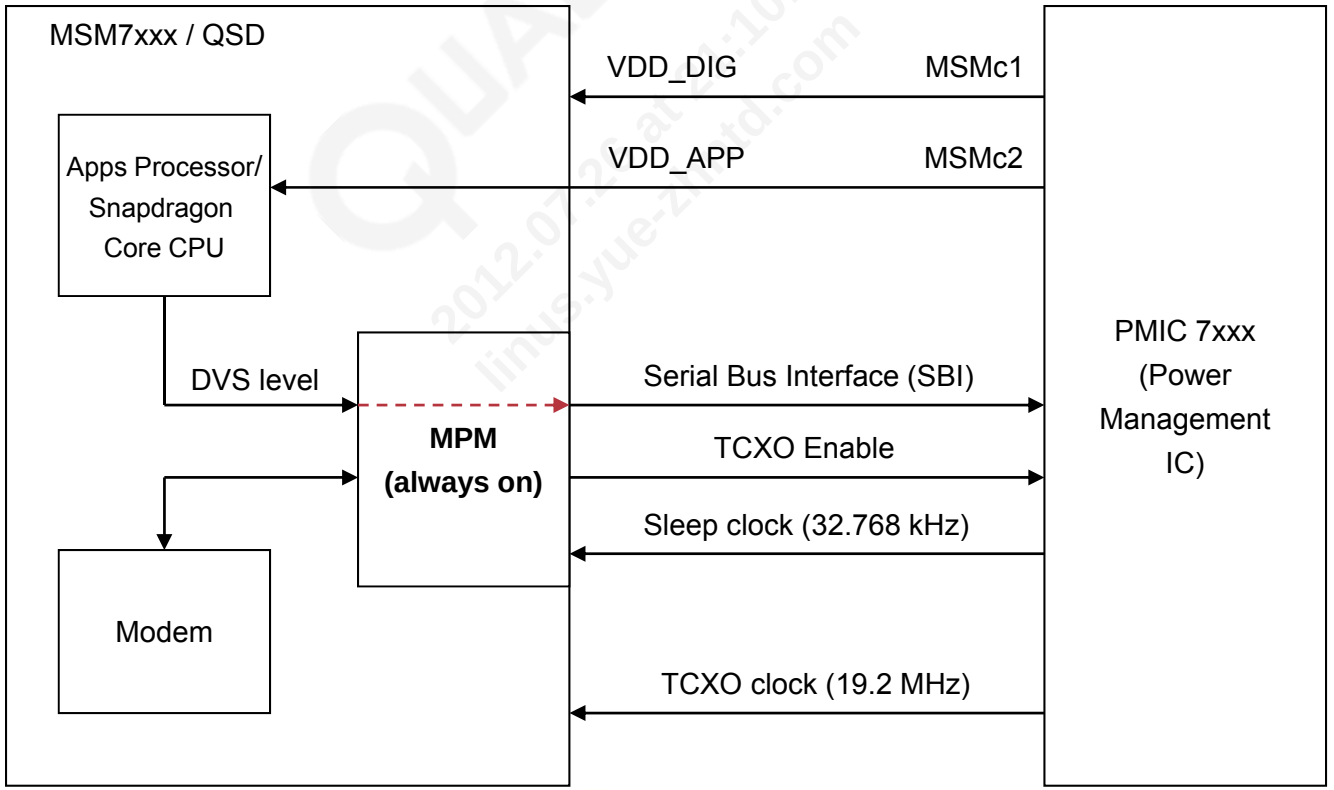
- Sleep thread runs as lowest priority
- Determines appropriate power-saving feature to use based on
  - Voting mechanism from other software components, including Apps Processor
  - Time remaining until next timer expiration
  - Time remaining until next paging channel slot
- Options
  - SWFI
    - Modem clock halted until interrupt detected
    - TCXO keeps toggling; all other enabled clocks keep toggling
  - TCXO shutdown
    - All clocks halted; SDRAM in Self-Refresh mode

Modem – Sleep Decision



Modem – MPM

- MSM disables/enables TCXO clock during sleep via MPM
- MSM sends commands to PMIC via MPM to change voltage configuration during sleep
- Apps Processor sends commands to PMIC via SBI to change MSMc2 (VDD_APP) voltage level for DVS



Modem – TCXO Shutdown

- Modem uses MPM to disable external TCXO until wake-up timeout expires or interrupt is detected by MPM
- Apps Processor shall be in Power-Collapsed (vote for TCXO shutdown) state as prerequisite
- Modem programs MPM as follows:
  - Sleep timeout at MPM (slow sleep clock)
  - Reroutes relevant interrupts to MPM as wake-up triggers
  - PMIC commands to lower voltage during sleep
- MPM communicates with PMIC to disable TCXO and reduce voltage to MSM digital core
- All clocks halted during TCXO shutdown period

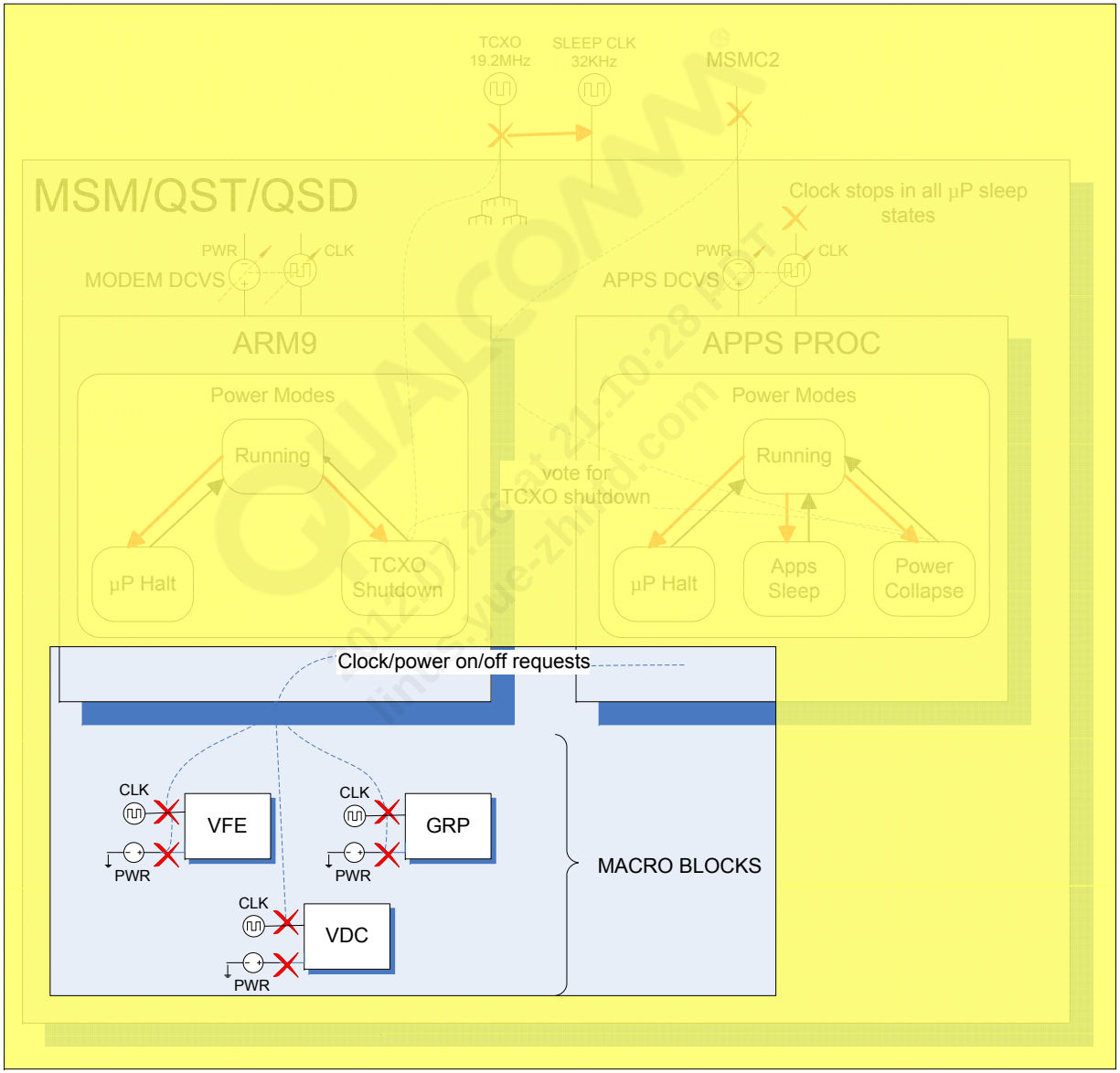
Modem – Enter TCXO Shutdown Procedure

- Configures MSM I/O to Power-Saving state
- Programs MPM with wake-up timeout
- Saves state of all clocks
- Disables all modem and peripheral clocks
- Reroutes interrupts to MPM
- Copies code into internal RAM
- Puts SDRAM in Self-Refresh mode
- Switches modem clock to TCXO
- Turns off PLL
- Executes SWFI to trigger MPM state machine
  - MPM disables external TCXO and sends command to PMIC to lower MSM voltage
  - Waits for timer expiration or other interrupt

Modem – Exit TCXO Shutdown Procedure

- Interrupt detected or wake-up timeout expires
- MPM sends command to PMIC to increase MSM voltage again and enables external TCXO
- Enables PLL
- Switches modem clock to full speed
- Brings SDRAM out of Self-Refresh mode
- Restores modem and peripheral clocks state
- Restores MSM I/O states

Hard Macro Power Collapse



Modem – Hard Macro Power Collapse

- Collapsible hard macros are VFE, VDC, and GRP
- Hard macros power-collapsed when not in use by driver/application
- Internal power source to MSM (no external interaction required)
- Turned on/off by software when corresponding clock is enabled/disabled
  - By using clock device driver API to enable/disable clocks
  - Modem clock device driver controls power for each hard macro
- Transparent from driver/application using hard macro
- Power-collapse is stateless, so driver/application must reinitialize corresponding hardware
- Hard macros turned on during chip reset, but immediately turned off by clock device driver initialization procedure

References

Ref.	Document	
Qualcomm		
Q1	*Application Note: Software Glossary for Customers*	CL93-V3077-1
Q2	*Linux Power Management Debugging Guide*	80-VR629-1 A

Questions?

https://support.cdmatech.com