MARVELL®

APPLICATION NOTE

Marvell® Scalable Power Manager

AN-xxx

Introduction

Marvell® Scalable Power Manager implements Scalable Power Manager in software. The power manager software is an open solution that can be used with all operating systems. The power manager software provides an infrastructure that uses power (voltage) and performance (frequency) scalability to fine-tune the power policy of a system.

Original equipment manufacturers (OEMs) and original design manufacturers (ODMs) can use the power manager software to manage power consumption and optimize system standby time and talk time in smart phones and PDAs that use the PXA27x processor. The PXA27x processor offers multiple operating modes and low-power modes as well as dynamic voltage management (DVM) and dynamic frequency management (DFM) capabilities. The power manager software optimizes the use of these modes and capabilities to dynamically scale power (voltage) and performance (frequency) under different workloads and idle conditions. The power manager software is an integral part of Marvell's Board Support Packages (BSPs) for the PXA27x processor.

The power manager software uses the native power management features and services that are provided by an operating system. The power manager software builds upon and extends these features and services by providing complementary applications programming interfaces (APIs), device programming interfaces (DPIs), services, policies and an infrastructure to increase power savings by intelligently managing power. Operating system vendors (OSVs) do not need to modify their operating systems in order to use the power manager software.

The power manager software provides device programming interfaces (DPIs) to device drivers and and Applications Programming Interfaces (APIs) to applications. The platform specific layer of the power manager software is used to adapt the power manager software to a given platform.

Users must design platforms to meet the requirements of the power manager software. Each device driver must be set up as a client of the power manager software so that the device receives notifications on power policy changes and power states via the DPIs. Optionally, applications can use the APIs to further enhance power savings.

Usage Modes

The power manager software has these usage modes:

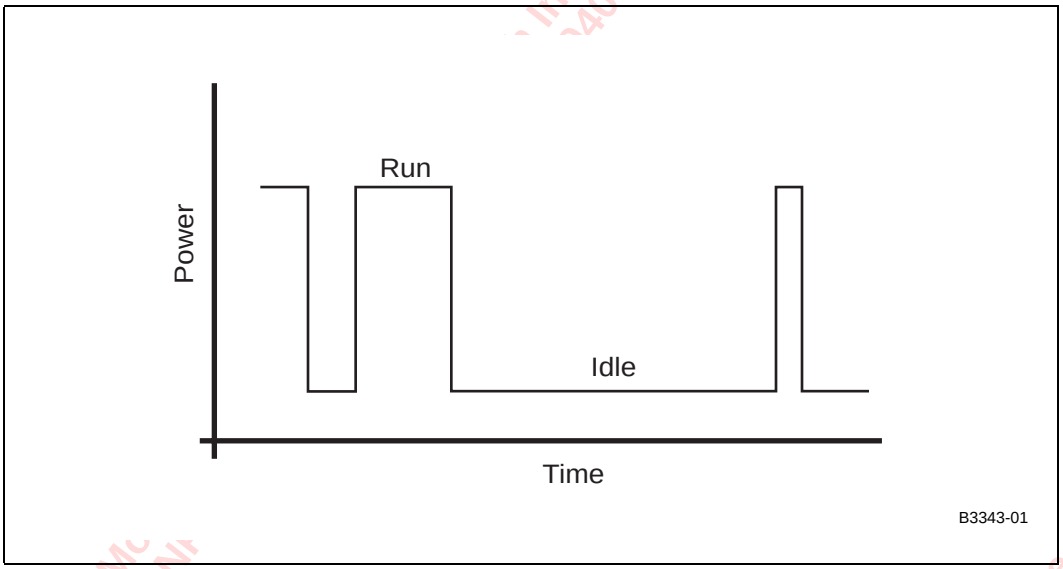
- Standby Mode (standby time)
- Voice Communications (talk time)
- Data Communications
- Multimedia (Audio, Video and Camera)
- Multimedia and Data Communications (Video Conferencing)

For each of these usage modes, the power manager software provides:

- Optimal power policy for dynamic scaling of power and performance
- Optimal operating frequency and voltage
- Usage of low power modes for the entire system including all of the devices
- State transition and power management for its devices

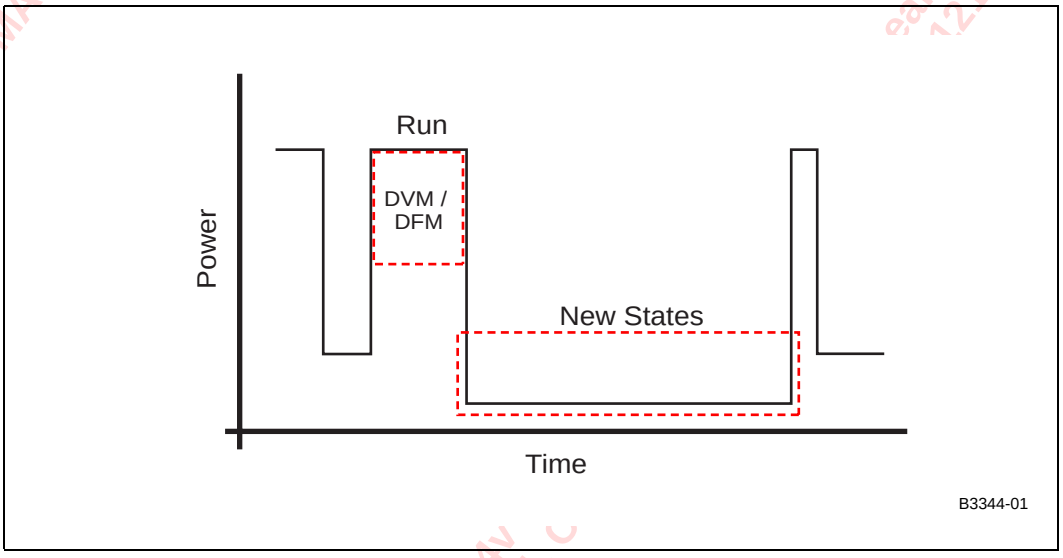
The operating profile of a generic system is characterized by a RUN/IDLE duty cycle as shown in Figure 1.

Figure 1: System Profile without Power Manager Software



The power manager software changes the operating profile of a generic system as shown in Figure 2. DVM and DFM are used to dynamically scale the “Run” frequency and voltage to meet immediate performance requirements with minimum power consumption. New power states are used to minimize “Idle” power consumption.

Figure 2: System Profile with Power Manager Software

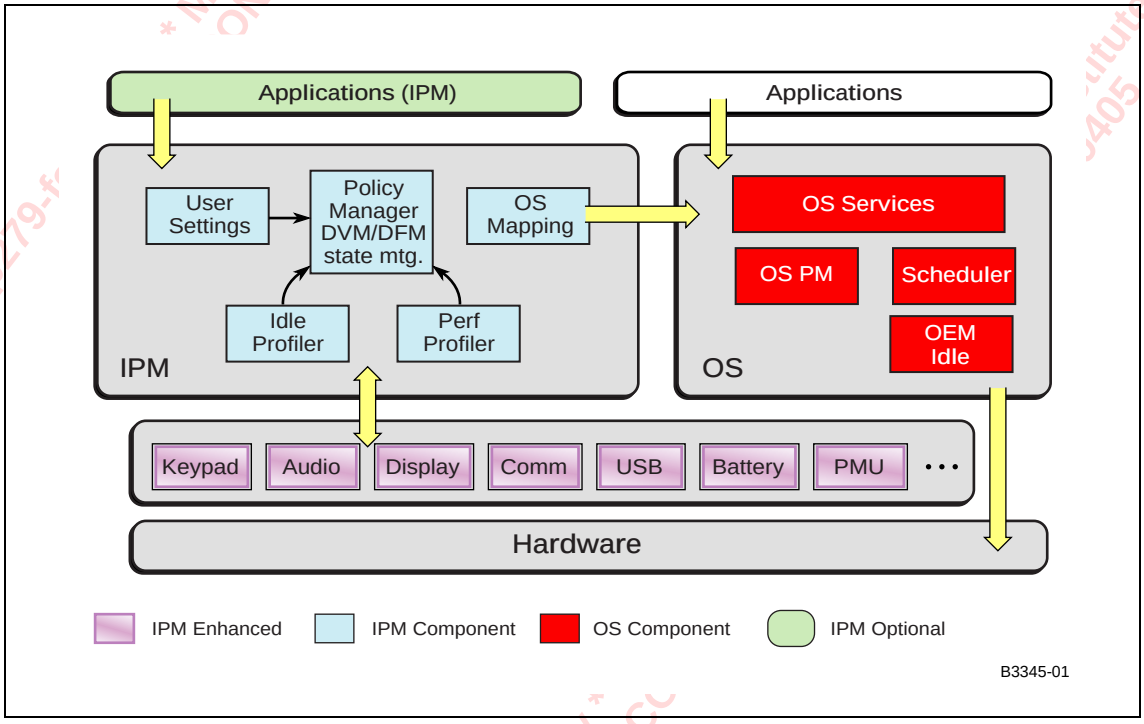


Architecture

The power manager software architecture consists of five software components as shown in Figure 3:

- **Policy Manager**: The Policy Manager is responsible for determining a system's power policy. The Policy Manager uses dynamic scaling of frequencies and voltages to provide the lowest power consumption under all types of workloads. The workloads may be processor bound, memory bound, or both processor bound and memory bound. The Policy Manager assesses information supplied by the Idle Profiler, Performance Profiler, User Settings, DPIs, and (optionally) APIs. The Policy Manager defines power states that run the PXA27x processor at the frequencies and voltages that are consistent with the lowest power consumption. If a power state is native to the operating system, the Policy Manager uses the native power state and the operating system's interface to the DPIs to transition the system into the specified power state. If the power state is not native to the operating system, the Policy Manager creates the power state and uses the DPIs to transition the system into the specified power state.
- **Idle Profiler**: Based on the workload, the Idle Profiler monitors parameters that are available in the operating system's Idle thread. The Idle Profiler then provides status information to the Policy Manager.
- **Performance Profiler**: The Performance Profiler uses the Performance Monitoring Unit (PMU) within the PXA27x processor to determine if the workload is CPU bound, memory bound, or both CPU and memory bound. The Performance Profiler then provides status information to the Policy Manager.
- **User Settings**: The User Settings allow the user to specify the parameters that are used by the Policy Manager to determine a system's power policy
- **Operating System Mapping**: The Operating System Mapping allows the power manager software to be ported across multiple operating systems.

Figure 3: Power Manager Software Architecture



Device Programming Interface

Each device driver uses its device programming interface (DPI) to register itself with the power manager software. The Policy Manager determines the best power state that meet system performance requirements with minimum power consumption. If the power state is native to the operating system, the power manager software uses the operating system's interface to notify the

device drivers of needed power state transitions within the devices. If the power state is not native to the operating system, the power manager software uses the DPIs to notify the device drivers of needed power state transitions within the devices. Power state notifications to the device drivers can be as simple as: enable the clock to Device #1 and then enable Device #1, and/or disable Device #2 and then disable the clock to Device #2. The device drivers must transition the devices into the specified power state and prepare the devices for the next event.

The Policy Manager also uses the DPI to communicate a new frequency and a new voltage (that meets system performance requirements with minimum power consumption) to the PXA27x processor power I^2C (PWR_I2C) device driver. The PWR_I2C unit within the PXA27x processor uses DFM and DVM to control the physical change to the specified frequency and voltage. Each device driver has the flexibility to request that the Policy Manger change the power state, e.g., the battery driver can monitor battery thresholds and then use its DPI to notify the Policy Manager of a needed power state change. Each device driver must be designed to be compatible with the power manager software.

Applications Programming Interface

Applications that have a dependency on performance can use the applications programming interfaces (API) to request the policy manager to quickly determine a frequency and a voltage that meets immediate performance needs with minimum power consumption. The Policy Manager uses the DPI to communicate a needed frequency and voltage to the Power I2C (PWR_I2C) device driver. The PWR_I2C Unit within the PXA27x processor uses DFM and DVM to control the physical change to the needed frequency and voltage. The power savings that are realized by applications that are enhanced (optional) to use the power manager software supplement the power savings that are realized with the help of the Idle Profiler, Performance Profiler, and DPIs.

Cellular Processor Interface

Communications software, shown on the right side of Figure 4, runs on the PXA27x processor. The power management component of the communications software manages power for the communications subsystem and it maintains its own state machine. This power management component interfaces to the L1/L2/L3 layers of the communications protocol stack and it has specific run/low power mode duty cycles associated with each of its states for both GSM and GPRS.

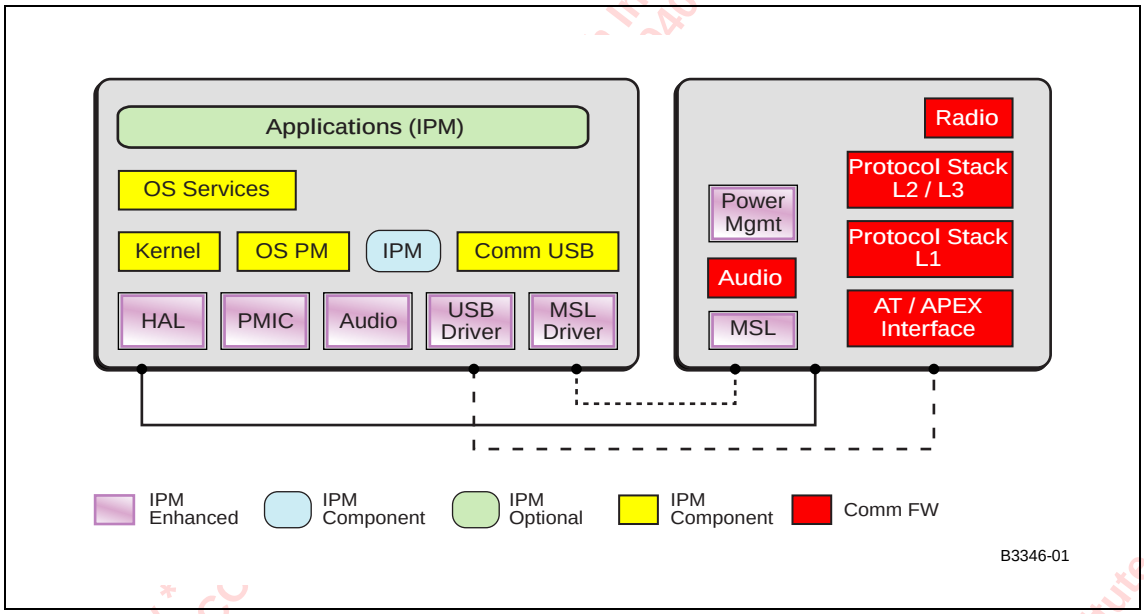
A physical link connects the application subsystem with the communications subsystem. An Mobile Scalable Link (MSL) can connect the PXA27x processor in the applications subsystem with a wireless cellular processor in the communications subsystem. Alternate physical links could use a UART or a synchronous serial port.

Communications device drivers that run on the applications subsystem are clients of the power manager software and the operating system's native power management component. These communications device drivers receive notifications from the power manager software and operating system's power management component on needed power state transitions. For example, when the OS goes into the Standby power state, these steps are performed:

- The power manager software notifies the communications driver of the need to transition the communications subsystem into the Standby power state.
- The communications driver sends a "change to Standby power state" message over the MSL to the communications power management component of the communications subsystem.
- The communications subsystem enters the Standby power state and prepares itself to wake-up the applications subsystem if the communications subsystem transitions into a new state that requires processing in the applications subsystem.

Similarly, for dynamic performance and power scaling, the communications device driver is notified about a frequency and voltage change, and in turn notifies the communications software of such via the MSL.

Figure 4: Communications Subsystem



PXA27x processor

The PXA27x processor implements Scalable Power Manager in hardware by providing:

- Reset sources
- Clock gating for peripherals
- Power modes

Reset Sources

- Power-on Reset
- Hardware Reset
- Watchdog Timer Reset
- GPIO Reset
- Reset on exit from Sleep mode
- Reset on exit from Deep-Sleep mode

Clock Gating for Peripherals

The PXA27x processor not only allows each of its peripherals to be independently enabled or disabled, it also allows the clock to each peripheral to be independently gated on or off.

Power Modes

The power modes:

- **Normal mode**: all internal power domains and external power supplies are fully powered and functional. The processor clocks are running.
- **Idle mode**: Clocks to the CPU are disabled; recovery is through interrupt assertion.
- **Deep-idle mode**: This mode can be entered only after the core frequency has been changed to 13 MHz. Clocks to the CPU are disabled; recovery is through interrupt assertion.
- **Standby mode**: All internal power domains except VCC_RTC and VCC_OSC are placed in a low-power mode where state is retained but no activity is allowed. The clock sources may be disabled. Some of the internal power domains can be powered off, and both PLLs are disabled. Recovery is through external and selected internal wake-up events.
- **Sleep mode**: All internal power domains except VCC_RTC and VCC_OSC (both are internal supplies) can be powered off. All clock sources, except those used by the real-time clock (RTC) and the power manager unit, are disabled. Software may optionally disable the external low-voltage power domains by de-asserting the PWR_EN pin. The remaining power domains are placed in a low-power state where state is retained but no activity is allowed. Recovery is through external and selected internal wake-up events. Because the program counter is invalid, recovery requires a system reboot (the program counter restarts from 0x0, so the core begins execution starting at the reset vector).
- **Deep-sleep mode**: All internal power domains except VCC_RTC and VCC_OSC can be powered off. All clock sources, except those used by the RTC and the power manager unit, are disabled. Software may optionally disable the external low-voltage power supplies. All power domains are powered directly from the backup battery pin, VCC_BATT. The remaining power domains are placed in a low power state where state is retained but no activity is allowed. Recovery is through external and selected internal wake-up events. Because the program counter is invalid, recovery requires a system reboot (the program counter restarts from 0x0, so the core begins execution starting at the reset vector).

The PXA27x processor provides 37 GPIO inputs that can be configured as Wakeup events to cause exit from the PXA27x processor low-power modes (all of the above modes except Normal are low power modes). For example, the PXA27x processor has several GPIOs that can act as Wakeup events when a key is depressed on a Keypad. This wakeup event can awaken the PXA27x processor from either the Standby or Sleep low power modes. Wakeup events could be caused by the following:

- Keypress
- Incoming Voice Phone Call
- Incoming GPRS Data
- Incoming SMS Message
- Headset Insertion
- USB Cable Insertion and Removal
- MMC Insertion and Removal
- RTC Wakeup
- Touchscreen activity
- Shell opening of a “Clam-Shell” phone

Programmable Frequency and Voltage Change Management

Programmable Frequency Change Management

The PXA27x processor core and peripheral clocks are derived from PLLs. The PXA27x processor implements programmable frequency change management (DFM) by allowing the core clock to be configured dynamically by software. The core clock frequency can be changed in several ways:

- Selecting the 13-MHz clock source
- Changing the core PLL frequency
- Enabling or disabling turbo mode or half turbo mode

Software programs the PXA27x processor Core Clock Configuration register (CCCR) to select:

- **Run Mode to Oscillator Ratio, CCCR[L] —** The L-bit determines the run frequency by multiplying the external crystal oscillator input by L.
- **Turbo Mode to Run Mode Ratio, CCCR[2N] —** The N-bit determines the turbo frequency by multiplying the run frequency by 2N.
- **Alternate Memory Controller Clock Selection, CCCR[A] —** If the A-bit is set, the memory controller's clock frequency is the same as that of the system bus; if the A-bit is clear, the memory controller's clock frequency is as shown in Table 1.

Software then programs Coprocessor 14, register C6 (CLKCFG) to select:

- **CLKCFG[B] — Fast Bus Mode**. If the B-bit is set, the system-bus frequency is equal to the run-mode frequency indicated in CCCR. When the B-bit is cleared, the system-bus frequency is equal to half the run-mode frequency indicated in the CCCR.
- **CLKCFG[F] —Core Frequency Change**. If the F-bit is set, the core PLL is stopped, and then restarted with the new CCCR settings.
- **CLKCFG[T] — Turbo Mode**. If the T-bit is set, the CPU operates at the turbo frequency; when the T-bit is cleared, the CPU operates at the run-mode frequency.
- **CLKCFG[HT] — Half-Turbo Mode**. If the HT-bit is set, whether the T-bit is set or clear, the CPU operates at the turbo frequency divided by two; when the HT-bit is clear, and the T-bit is clear, the CPU operates at the run-mode frequency; when HT is clear, and T is set, the CPU operates at the turbo frequency.

An automatic frequency change sequence is initiated when software sets CLKCFG[F]. This sequence establishes the values in CCCR and the selected modes in CLKCFG.

Programmable Voltage Change Management

The PXA27x processor implements programmable voltage change management (DVM) through its voltage manager. The voltage manager provides voltage management through use of an I^2C unit (PWR_I2C) that is dedicated to communication with an external PMIC regulator, and through use of a voltage change sequencer.

When software initiates a voltage-change mode, the voltage change sequencer can automatically send commands via the PWR_I2C unit to an external PMIC regulator. The sequencer can send up to 32 commands, which can be categorized as dynamic commands and static commands:

- ***Dynamic commands*** are executed when the core is running.
- ***Static commands*** are executed after clocks to the processor are disabled.

The PXA27x processor uses its Power Manager General Configuration register (PCFR), Power Manager Voltage Change Control register (PVCR), and Power Manager I^2C Command register File (PCMDx) to define and control a voltage change sequence:

- **Frequency/Voltage Change, PCFR[FVC]** — If the FVC bit is set, a frequency change sequence (DFM) also triggers a voltage change sequence (DVM).
- **Read Pointer, PVCR[RP]** — The read pointer field in the PVCR points to the PCMD register location that contains a command or the first command of multiple commands that are to be sent to the external regulator via the PWR_I2C. The command sequence can start from any one of 32 PCMD registers. After a command is sent out, the Read Pointer increments to point to the next PCMD register location. The read pointer is not incremented if the current command is the last command, as indicated by PCMD[LC] set.
- **Delay Command Execution, PCMD[DCE]** — If the DCE bit is set in the current PCMD, a counter (set by the command delay bits in PVCR) waits for a programmable number of 13-MHz processor-oscillator cycles before continuing execution of the command. This is useful if a longer period between commands is required.
- **Multi-Byte Command, PCMD[MBC]** — If the MBC bit is set, the voltage-change sequencer continues sending bytes with no delay or handshaking with the power manager unit until a command with the MBC bit clear is executed.
- **Last Command, PCMD[LC]** — If the LC bit is clear, the voltage-change sequencer expects the PCMD register at the next higher address to contain an additional command. If the LC] bit is clear in PCMD31, the PVCR Read Pointer rolls over to PCMD0 after executing the command in PCMD31. When the LC bit is set, the Voltage Change Sequencer considers the

current command to be the last one and finishes after execution completes. Each voltage change command sequence must be terminated by setting the LC bit of the last command in the sequence. The PVCR Read Pointer is not incremented if the LC bit is set.

Coupling Voltage Change with Frequency Change

A frequency change (clock source change or core PLL frequency change) may be used to change the frequency of the core, system bus, memory controller, and LCD controller to a value not available with turbo or fast-bus modes. This frequency change can be coupled with a voltage change by setting PCFR[FVC]. Similarly, a voltage change can be coupled with a change to or from fast-bus mode.

Programmable Operating Frequencies

The PXA27x processor programmable operating frequencies are shown in Table 1.

Table 1: Clock Frequencies

Core Run Freq	CLKCFG[T]	Core Turbo Freq	CLKCFG[T]	CLKCFG[HT]	CCCR[L]	CCCR[2N]	System Bus	CLKCFG[B]	CLK_MEM (Memory Controller)	CCCR[A]	SDCLK<2:1> SDRAM Clocks	MDREFR[KxDB2]††	Synchronous Flash	MDREFR[K0DB4]	MDREFR[K0DB2]	LCD
91.0†	0	—	—	0	7	2	45.0	0	91.0	0	45.0	1	22.5	1	1	91.0
104.0	0	104.0	1	0	8	2	104.0	1	104.0	1	104.0	0	52.0	0	1	52.0
156.0	0	156.0	1	1	8	6	104.0	1	104.0	1	104.0	0	52.0	0	1	52.0
104.0	0	312.0	1	0	8	6	104.0	1	104.0	1	104.0	0	52.0	0	1	52.0
208.0	0	208.0	1	0	16	2	208.0	1	208.0	1	104.0	1	52.0	1	X	104.0
208.0	0	312.0	1	0	16	3	208.0	1	208.0	1	104.0	1	52.0	1	X	104.0
208.0	0	416.0	1	0	16	4	208.0	1	208.0	1	104.0	1	52.0	1	X	104.0
208.0	0	520.0	1	0	16	5	208.0	1	208.0	1	104.0	1	52.0	1	X	104.0
208.0	0	624.0	1	0	16	6	208.0	1	208.0	1	104.0	1	52.0	1	X	104.0

NOTES:

† L=7 (Core = 91.0 MHz) must be strictly used as the bootup frequency and immediately reconfigured to one of the other frequency points.

†† KxDB2 represents K1DB2 and K2DB2

Workload Characterization for DFM and DVM

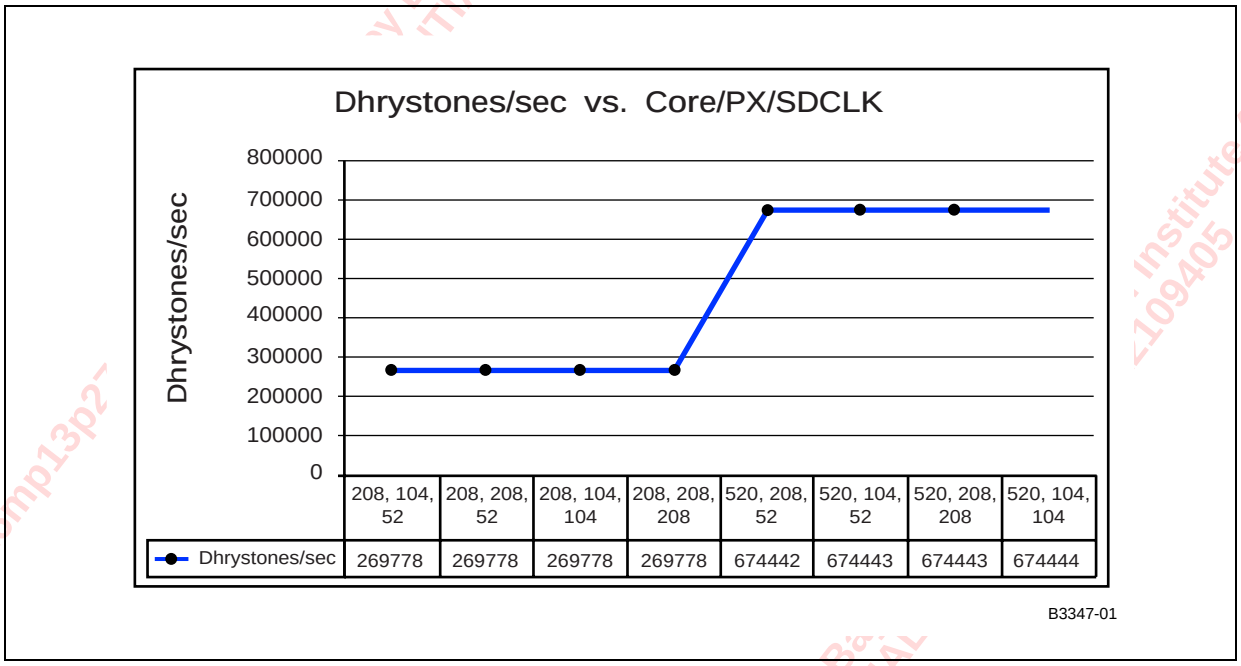
Most software applications/workloads can be generalized into three main categories:

- CPU (compute) bound applications
- Memory bound applications
- I/O bound applications

CPU Bound Applications

An application is generally considered CPU-bound when most of its execution time is spent on computation, using the data and instructions loaded in D-cache and I-cache. Scheduling instructions suitable to the underlying processor architecture (reducing stalls) can potentially increase performance of CPU-bound applications. These applications tend to keep the CPU busy all of the time and the processor's idle time is negligible. An increase in processor frequency (and in turn voltage) helps to increase the performance of these applications. As an example, Dhrystone is a purely CPU bound workload and as shown in Figure 5, the performance is a linear function of CPU (core) frequency. In order to meet performance requirements, it is essential to have these applications run at the maximum possible frequency. This information is very critical for a performance-optimizing policy manager.

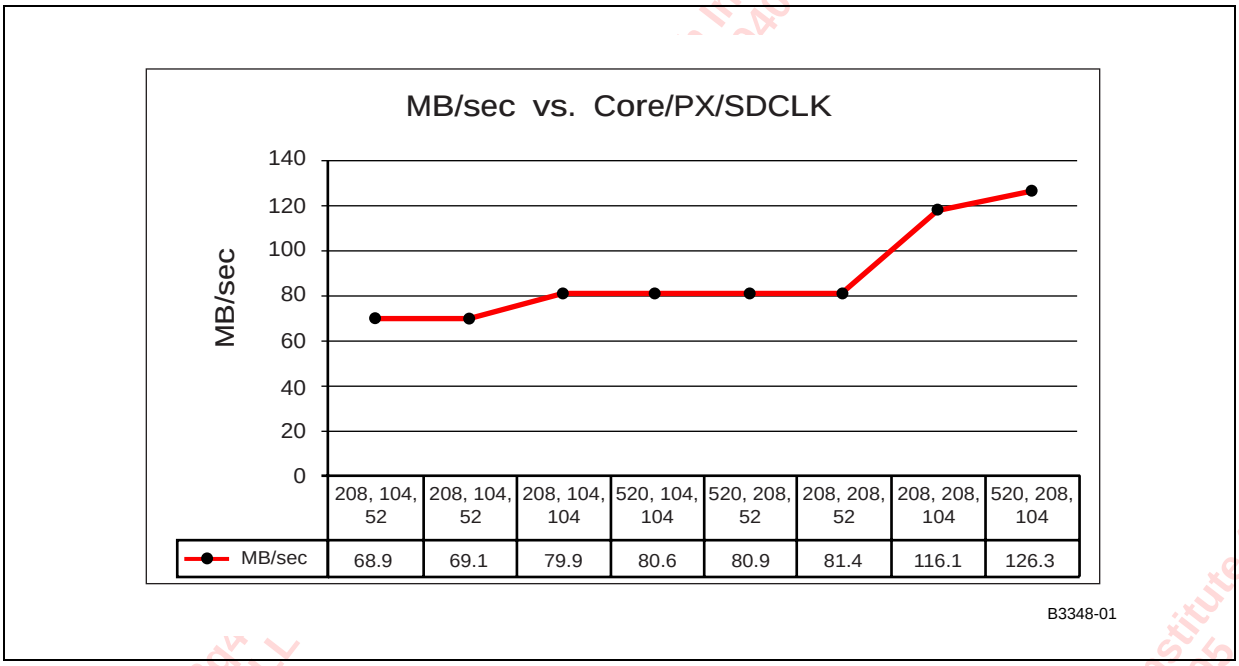
Figure 5: Dhrystone vs. CPU/Core Frequency



Memory Bound Applications

Some applications that work on large data blocks (greater than the cache size) usually have to access data outside of the caches, and become bound by the memory or the system bus speed. These applications, such as a memory copy, move large blocks of data and tend to generate significant memory traffic, with most CPU cycles lost waiting for data. In such cases, performance does not improve (see Figure 6) even if the core's speed is increased, since the performance is a function of the memory speed. This information is very critical for a performance-optimizing policy manager.

Figure 6: Memory vs. CPU/Core Frequency



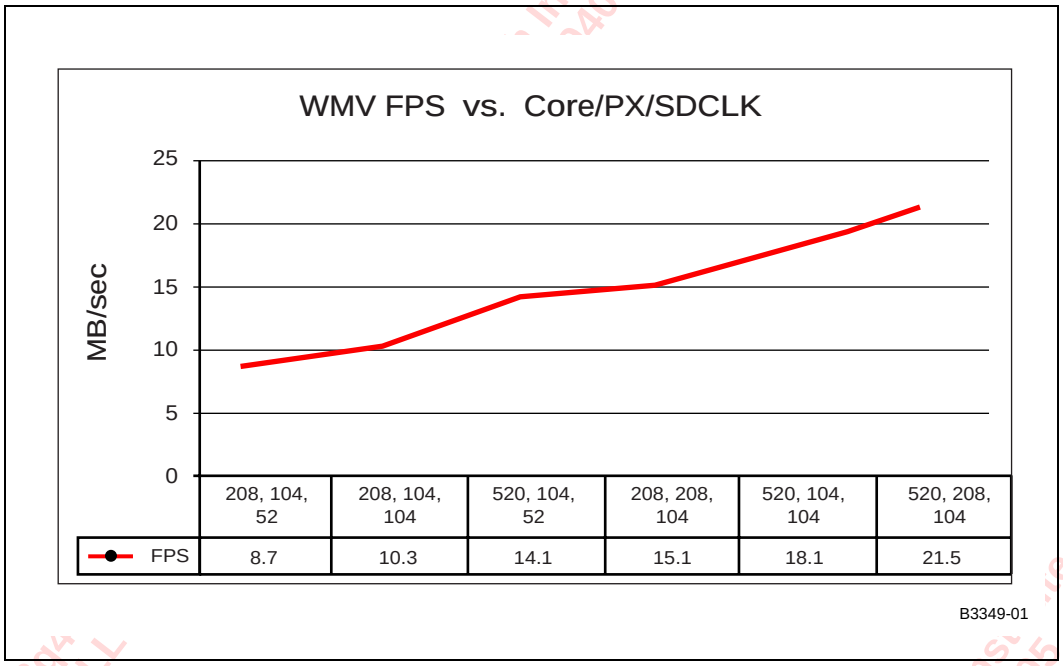
I/O Bound Applications

Applications that are waiting on some I/O (peripheral) device for data are considered I/O bound. An example would be an Ethernet driver waiting for data from the network.

CPU and Memory Bound Applications

Many applications have performance demands that vary over time. At a given instant they could be either CPU (compute) bound or memory bound. Multimedia applications that undertake a large amount of computations as well as work on large data blocks fall in this category. The characteristics of these multimedia applications show that performance is bounded by both memory and CPU speed. For these types of workloads, accurate prediction or estimation of the characteristics yields a better power policy. For example, a video player is a CPU and memory bound type of application. Its performance is plotted as a function of the core and memory frequency in Figure 7.

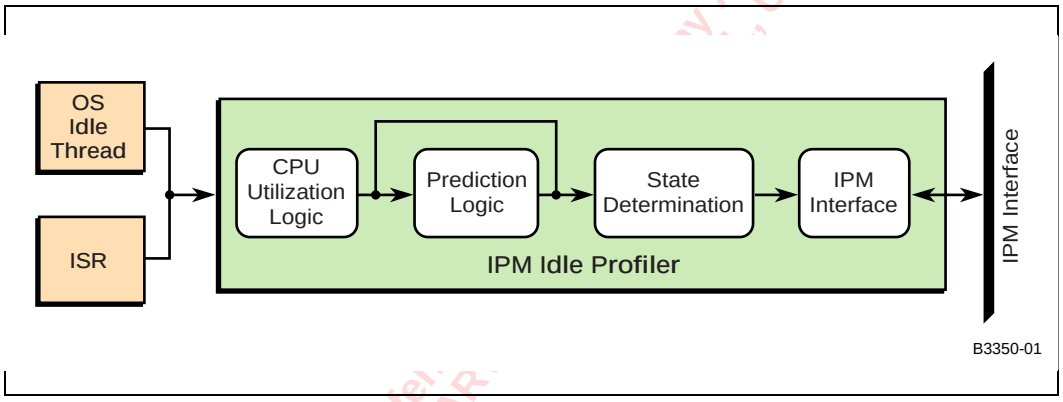
Figure 7: Performance vs. CPU/Core Frequency



Idle Profiler

The Idle Profiler provides CPU usage and operating system idle information to the power manager software. Figure 8 shows the operating system's idle thread providing input to the Idle profiler. Since the idle thread is executed when the OS is not busy (not executing any code), it is one of the preferred choices to provide CPU usage information. However, since the idle thread only executes when there are no tasks ready to run, the information is only provided when the CPU is used less than 100% of the time. In cases where CPU usage is less than 100% of the time, but still very high, the ISR can be used to provide CPU usage information to the Idle Profiler.

Figure 8: Idle Profiler, Power Manager Software

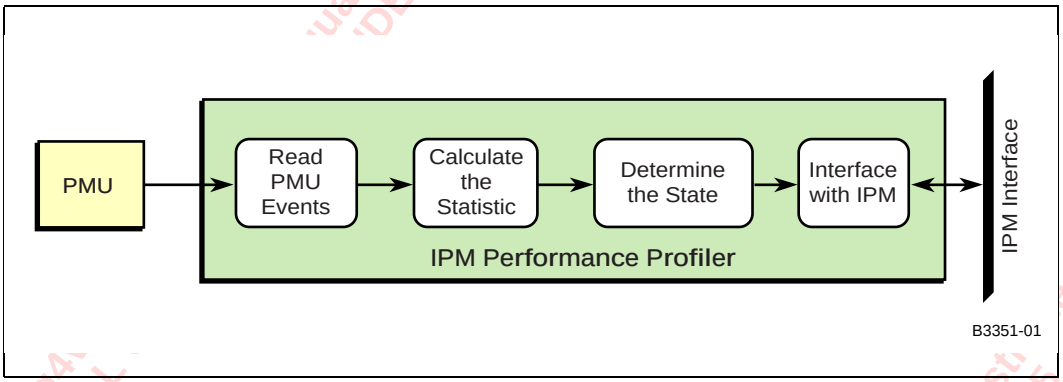


Performance Profiler

System workload does not remain static at any given time. So, dynamic workload characterization is essential to the optimization of system performance at minimum power dissipation. The Performance Profiler monitors system information and maintains a system state. At any given time, the Policy Manager can direct the Performance Profiler to return the current system state. Since the Performance Profiler is event driven, it can automatically alert the Policy Manager when the system state changes.

In order to achieve dynamic scaling for power and performance based on dynamic characterization of CPU bound, memory bound, or CPU and memory bound workloads, the Performance Profiler monitors the PXA27x processor Performance Monitoring Unit (PMU) as shown in Figure 9.

Figure 9: Performance Profiler



The PMU tracks processor events as described in Table 2. The Performance Profiler also monitors the status registers of peripheral devices such as the PXA27x processor LCD Controller and monitors the PXA27x processor DMA activity. This monitoring allows the Performance Profiler to dynamically characterize the system state.

Table 2: PXA27x processor Performance Monitoring Unit (PMU)

Information Source	Description of Event	Usage for Characterization
PMU Registers	Instruction cache-miss	Instruction traffic and CPI estimation and cache locality
	Instruction cache cannot deliver instruction	
	Data dependency stall	%Memory use
	Instruction TLB miss	Page locality
	Data TLB miss	
	Instruction executed	%CPU Used
	Number of Stalls an number of times the stall occurs due to D-Cache buffer full (every cycle condition is present)	Data traffic / Data access congestion / Congestion rate and congestion lengths
	Data cache access	Data access rate, memory bound %Memory use
	Data cache-miss	Data traffic and %Memory use
	Data cache write-back	Data traffic
	Software changed the PC	Number of function calls
	DMA status on number of descriptors	DMA traffic
	Peripheral status of buffer	Peripheral traffic

Policy Manager

The policy manager defines power states and maps them to power modes of the PXA27x processor. If a power state is native to the operating system, the policy manager uses the native power state and the operating system's interface to the DPIs to transition the system into the specified power state. If the power state is not native to the operating system, the Policy Manager creates the power state and uses the DPIs to transition the system into the specified power state.

The Policy Manager uses device driver inputs, application workload inputs (optional), Idle Profiler inputs, and Performance Profiler (system workload) inputs to define the system's power policy. Alternatively, an OEM can use these inputs to define its own power policy.

The power manager software provides an infrastructure to the device drivers for state changes, frequency changes, and voltage changes. Using this infrastructure, the power manager software notifies the client drivers of these changes either directly or through the operating system's power management services.

A system that is enabled by the power manager software always maintains a predefined operating point as part of a global variable, namely

IPMOperatingPoint = [State, Voltage, Frequency, Frequency2, Frequency3]

Where:

- State = processor state
- Voltage = processor core voltage
- Frequency1 = processor core frequency
- Frequency2 = system bus frequency
- Frequency3 = SDCLK frequency

For example, the state of the PXA27x processor could be run, sleep, standby, or 13M.

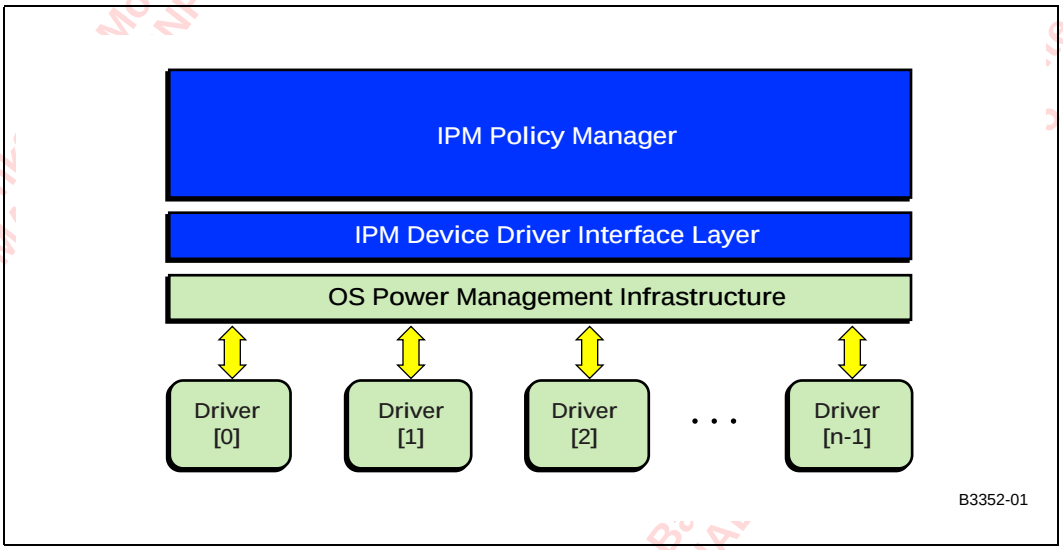
Similarly, the voltage and frequency can take on any of the values supported by the PXA27x processor. At any given instant, the output of the Policy Manager sets IPMOperatingPoint to the optimum values that achieve the required performance at the lowest power dissipation.

Device Drivers Interface

Overview

The power manager software device interface layer works with the device drivers and the power manager software policy manager as shown in Figure 10.

Figure 10: Driver Drivers Interface, Power Manager Software



Since many operating systems have some level of native power management infrastructure, the power manager software interfaces with this native infrastructure wherever possible. If the operating system does not have a native power management infrastructure, the power manager software adds management infrastructure The power manager software ports the Driver Interface Layer of each operating system, providing a level of abstraction that allows the Policy Manager to be de-coupled from direct driver interaction.

The following sections describe the generic architecture of Driver interaction. The implementation details differ over various operating systems, and are addressed in separate OS-specific documentation.

Registration

The power manager software has the ability to notify drivers when power parameters change. This notification allows a driver to prepare for the pending transition, or to veto the transition if necessary. During registration, the driver must supply a callback

function either to the existing infrastructure or directly to the power manager software if no such infrastructure exists. The driver also must inform the power manager software of the target device's frequency and state sensitivity so that the power manager software only notifies the driver of a pending state transition when necessary.

Device Driver Interface Layer

Based on the implicit direction of flow, there are two categories of Driver interface types:

- Power Manager Software**-to-Driver Interface**: This interface defines a set of routines that provide inputs to a given registered driver. These routines allow the power manager software to notify a registered driver of its intention to alter the system power state in some way, such as entering Standby mode or Idle mode. Drivers must be able to veto the power manager software request to perform a power parameter transition if doing so causes undesirable behavior.
- **Driver-to-**Power Manager Software **Interface**: This interface defines a set of routines that a registered driver can use to provide unique information regarding the target device's loading and power state to the power manager software. This information can help the power manager software make informed decisions in determining the optimal system power state. At a minimum, a registered driver might only provide a stub for these routines.

It is important that all drivers maintain their own device power state management. The drivers must at least be able to monitor their own activity and self-manage their own state, such as turning off the peripheral device when it makes sense and informing the power manager software of the transition. The power manager software tracks the state of each driver, but does not mandate the device's state management.

Device Power states

Since many operating systems currently have some level of device power management, the Device Driver Interface Layer translates an OS-specific device state into an appropriate power manager software-specific state. If an operating system does not supply power management device states, the device drivers use the power states created by the power manager software. When a registered driver transitions its device's power state, the driver must inform the power manager software of this transition. power manager software device states are shown in Table 3.

Table 3: Device States

State	Description
On	The device is fully powered up and fully functional.
Off	The device is off.
Low Power	The device is on, but is in a reduced power state. Depending on the device there could be several of these low power states.

Device State Sensitivity

As part of the driver's registration, the driver must report to the power manager software what the device's sensitivity is to all supported processor modes. If a driver reports that it is sensitive to a given state, the power manager software notifies the driver before the power manager software makes the state transition.

Device Frequency Sensitivity

As part of a driver's registration, the driver must inform the power manager software as to whether or not the driver requires notification when the Policy Manager intends to scale the device's frequency and/or voltage. For example, Table 1 shows that when the Policy Manager intends to scale the PXA27x processor core run frequency from 208 MHz to 104 MHz, other PXA27x processor frequencies (e.g., system bus, memory controller, LCD controller) must be scaled as well.

Power Manager Software-Aware Application Interface (Optional)

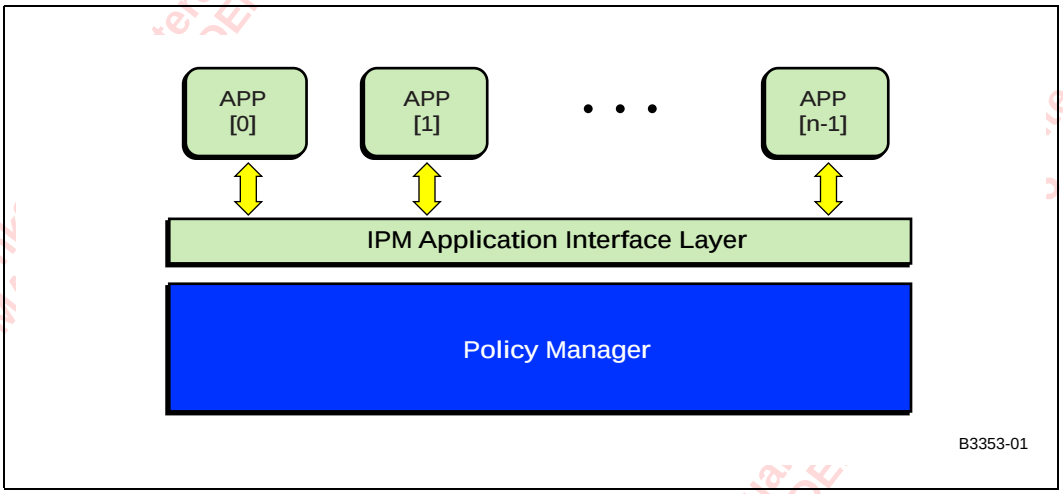
Overview

When applications are slightly enhanced to provide their performance/processing requirements (cycles, deadlines etc.), frequency change and voltage change scheduling becomes simpler and more effective. For example, consider a mobile platform that uses the PXA27x processor to control a small-sized display. For an MPEG4 video application to decode and display 30 fps, the PXA27x processor can run at its maximum possible frequency and voltage so that significant idle time exists during the execution bursts that are needed to service the application, as shown in Figure 2. The Idle Profiler alone would either not detect this idle time (if it is very small) or would incur a latency in detecting the idle time. Optionally, if the application is slightly enhanced to convey the idle time information to the Power Manager, the Policy Manage can make good decisions to reduce power dissipation during the idle time.

Application Programming Interface

The Applications Programming Interface (API) is shown in Figure 11.

Figure 11: Applications Programming Interface



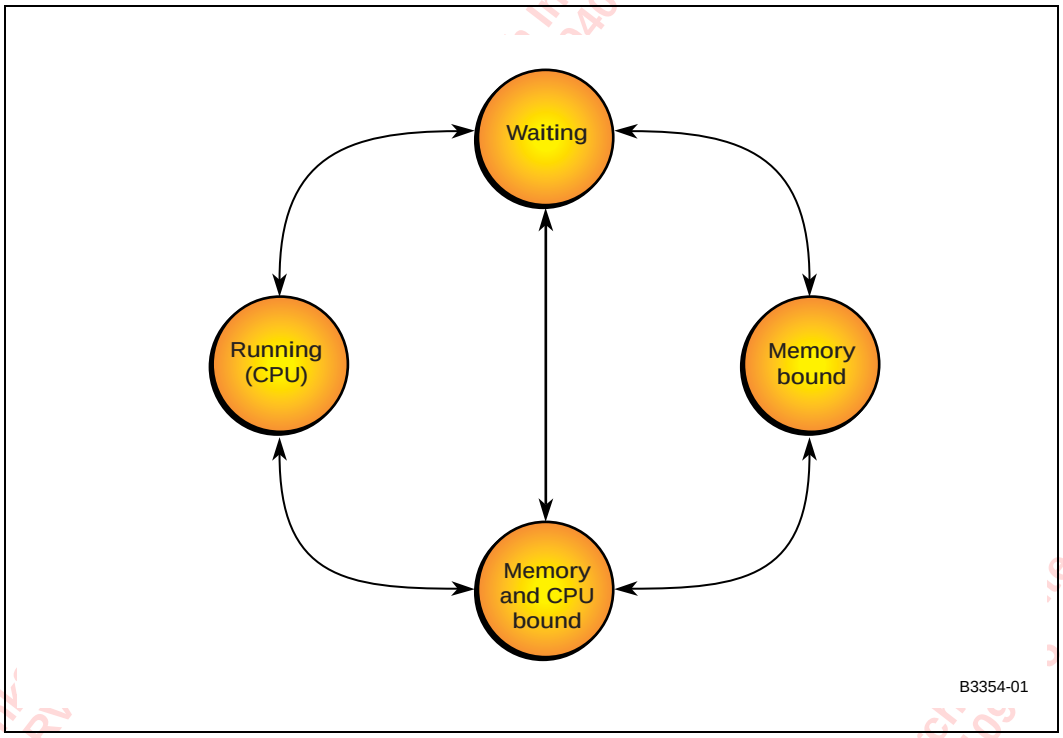
The API is designed to allow communication between the Policy Manager and each power manager software-aware application. Applications must register with the power manager software and provide power state information to the power manager software.

Application Power States

At any given instant, each application must be in one of four power states as shown in Figure 12:

- **Running** - This state is the most common power state. An application is running/executing instructions and all needed data is available, i.e., the application does not expect to have any data-stalls. A Dhrystone application is an example of this CPU bound power state.
- **Waiting** - An application is polling a peripheral and is waiting for a response, or an application is idling. An application that is waiting for network data is an example of this I/O bound power state.
- **Memory Bound** - An application is moving large blocks of data. A Memory-Copy (MEMCPY) operation is an example of this Memory bound power state.
- **Memory and CPU Bound** - An application is running a complex algorithm on blocks of memory data. A a video game is an example of this Memory and CPU Bound power state.

Figure 12: Application Power States



Programming Functions and Structures

Idle Profiler

Functions

These functions are used to communicate between the Policy Manager and the Idle Profiler. These functions are internal and are used between software components within the power manager software.

- IPMStatus IPMIdleProfilerInit(void) – Provides a function that the Policy Manager will call to initialize the Idle Profiler.
- IPMStatus IPMIdleProfilerExit(void) – Provides a function that the Policy Manager will use to cause the Idle Profiler to halt and clean up. The Policy Manager will not receive information from the Idle Profiler when it is disabled. Another call to IPMIdleProfilerInit() is required to restart the Idle Profiler.
- SystemState IPMIdleProfilerGetState() – Provides a function that the Policy Manager will use to query the Idle Profiler for the current system Idle state.

Structure

This data structure contains required information that is passed between the Policy Manager and the Idle Profiler:

```
struct IPMIdleInfo {
    int HiEdge;
    int LowEdge;
```

```
    int MedEdge;
    int CurEdge;
    int Utilization;
} IPM_IdleInfo;
```

Performance Profiler

Functions

- IPMPerfProfilerInit
  - *Prototype*: IPMStatus IPMPerfProfilerInit(IPMPMUInfo pmuinfo)
  - *Discussion*: A function that the Policy Manager calls to initialize the PMU. The initialization includes: 1) setting up the PMU to read desired events and 2) defining variables required for the operation of the Performance Profiler.
  - *Input Arguments*: `pmuinfo` – Pointer to the `IPMPMUInfo` structure
  - *Output Arguments*: None
  - *Return Values*:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

- IPMPerfProfilerExit
  - *Prototype*: IPMStatus IPMPerfProfilerExit(IPMPMUInfo pmuinfo)
  - *Discussion*: A function that the Policy Manager calls to do the de-initialization for the PMU. The programming interface implements appropriate commands to stop the PMU, and perform any needed clean-up.
  - *Input Arguments*: `pmuinfo` – Pointer to the `IPMPMUInfo` structure
  - Output Arguments: None
  - *Return Values*:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

- IPMPerfProfilerGetState
  - *Prototype*: IPMStatus IPMPerfProfilerGetState(IPMPMUInfo pmuinfo)
  - *Discussion*: A function that the Policy Manager calls to retrieve the current operating state from the Performance Profiler.
  - *Input Arguments*: `pmuinfo` – Pointer to the `IPMPMUInfo` structure
  - *Output Arguments*: None
  - Return Values:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

- IPMPerfProfilerUpdateState
  - *Prototype*: IPMStatus IPMPerfProfilerUpdateState(IPMPMUInfo pmuinfo)
  - *Discussion*: A function that the Performance Profiler calls when it detects a system state change. The programming interface provides the new policy to the Policy Manager through the common structure.
  - *Input Arguments*: `pmuinfo` – Pointer to the `IPMPMUInfo` structure
  - *Output Arguments*: None

  - *Return Values*:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

- IPMPerfProfilerGetCPUUtilization
  - *Prototype*: IPMStatus IPMPerfProfilerGetCPUUtilization(IPMPMUInfo pmuinfo)
  - *Discussion*: A function that the Performance Profiler calls when it detects a system state change. This programming interface provides the new policy to the Policy Manager through the common structure.
  - *Input Arguments*: `pmuinfo` – Pointer to the `IPMPMUInfo` structure
  - *Output Arguments*: None
  - *Return Values*:

    IPMNoErr – Indicates no error

    IPMErr – Indicates error

- IPMPerfProfilerGetMemEfficiency
  - *Prototype*: IPMStatus IPMPerfProfilerGetMemEfficiency(IPMPMUInfo pmuinfo)
  - *Discussion*: A function that the Performance Profiler calls when it detects a system state change. The programming interface provides the new policy to the Policy Manager through the common structure.
  - *Input Arguments*: `pmuinfo` – Pointer to the `IPMPMUInfo` structure
  - *Output Arguments*: None
  - Return Values:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

Structure

This data structure contains required information that is passed between the Policy Manager and the Performance Profiler. The structure is defined as:

```
typedef enum {
    cpuBound =0,   //System is CPU bound
    memoryBound=1,  //System is memory bound
    Idle=2         //System is idling
    } PerfState;
struct IPMPMUInfo {
    enum CurrentState;
    int CPUUtilization;
    int MemEfficiency;
            …..
} IPMPMUInfo;
```

Policy Manager

Private

This function is used for Policy Manager-to-IDDIL component communication.

- IPMEnterLPState
  - *Prototype*: BOOL IPMEnterLPState(IPMState LPSTATE);
  - *Discussion*: When the Policy Manager determines that the system power state needs to transition, it will call into the IDDIL. It is the job of the IDDIL to determine which drivers require notification and inform the Policy Manager if a driver has vetoed the transition.
  - *Input Arguments*: `LPSTATE` – The Policy Manager will populate this function with the operating parameter information that it wants to transition to and then pass them to the `IDDIL` to handle the OS-specific task of informing the drivers.
  - *Output Arguments*: None
  - *Return Values*:

    True – indicates the `IDDIL` has informed all interested drivers and no objections were received.

    False – indicates at least one driver has objected, thereby blocking the transition.

Public

These functions are used for communication between the Policy Manager and the drivers and/or the operating system's existing power management infrastructure.

- IPMBlockStateTransition
  - *Prototype*: IPMStatus IPMBlockStateTransition(DriverHandle);
  - *Discussion*: This function allows a driver to prevent the Policy Manager from making any state transitions, letting a driver bracket sensitive code sequences. Drivers must use this sparingly since there is no power saving in the device during a block. Drivers should use `IPMUnBlockStateTransition(DevID)` to remove the block.
  - *Input Arguments*: `DriverHandle` – This is the driver handle assigned to the device at the time of registration.
  - *Output Arguments*: None
  - *Return Values*:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

- IPMUnBlockStateTransition
  - *Prototype*: IPMStatus IPMUnBlockStateTransition(DriverHandle);
  - *Discussion*: This function allows a driver to de-assert its request that the Policy Manager make no system state transitions.
  - *Input Arguments*: `DriverHandle` – This is the driver handle assigned to the device at the time of registration.
  - *Output Arguments*: None
  - *Return Values*:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

- IPMRegisterDeviceDriver
  - *Prototype*: IPMStatus IPMRegisterDeviceDriver(*DriverHandle, (*CallBack)(…));

  - *Discussion*: This function allows a driver to register with the Policy Manager. The driver needs to provide a callback function for communication with the Policy Manager. The Policy Manager will assign a driver handle for further communications between itself and the driver. A driver should call `IPMAddDevice()` to add devices.
  - *Input Arguments*: `CallBack` – The registering driver needs to populate the callback to enable the Policy Manager to communicate state transitions.
  - *Output Arguments*: `DriverHandle` – The Policy Manager will assign each registered driver a driver handle for future communications.
  - Return Values:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

- IPMDeRegisterDeviceDriver
  - *Prototype*: IPMStatus IPMDeRegisterDeviceDriver(DriverHandle);
  - *Discussion*: This function lets a driver unregister from the Policy Manager. The Policy Manager, in turn, removes all data structures related to the unregistering driver.
  - *Input Arguments*: `DriverHandle` – The driver handle assigned at the time of registration.
  - *Output Arguments*: None
  - Return Values:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

- IPMGetOperatingFreqs
  - *Prototype*: IPMStatus IPMGetOperatingFreqs(DriverHandle, FreqArray, *NumElements);
  - *Discussion*: This function allows a registered driver to gain information regarding the set of frequencies at which the Policy Manager can operate.
  - *Input Arguments*: `DriverHandle` – Driver handle assigned by the Policy Manager at the time of registration.
  - *Output Arguments*:

    `FreqArray` – An array of FreqStruct filled by the Policy Manager at initialization.

    `NumElements` – Indicates how many elements are in FreqStruct.
  - *Return Values*: None

- IPMRegisterDriverSensitivity
  - Prototype: IPMStatus IPMRegisterDriverSensitivity(DriverHandle, DriverFreqSensitivity, DriverStateSensitvity);
  - *Discussion*: This function allows for a driver to register its sensitivity to both processor states and frequencies with the Policy Manager. The driver should simply set the corresponding bits in each bit field for which it is sensitive, and will require the Policy Manager to provide notification before making a transition.
  - *Input Arguments*: `DriverHandle` – Driver handle assigned by the Policy Manager at the time of registration.

    `DriverFreqSensitivity` – Bit field representing which FreqArray elements require driver notification.

    `DriverStateSensitivity` – Bit field representing which CPUState enumerations require driver notification.
  - Output Arguments: None
  - Return Values:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

- IPMAddDevice
  - *Prototype*: IPMStatus IPMAddDevice(*DeviceId, DriverHandle, DeviceState);
  - *Discussion*: A registered driver should call this function to add a device that it manages.
  - *Input Arguments*:

    `DriverHandle` – The driver handle assigned to the driver by the Policy Manager at the time of registration.

    `DeviceState` – Type `DevStatesEnum`, this represents the current state of the device.
  - *Output Arguments*: `DeviceID` – The device identifier assigned by the Policy Manager.
  - *Return Values*:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

- IPMRemoveDevice
  - *Prototype*: IPMStatus IPMRemoveDevice(DeviceId, DriverHandle);
  - *Discussion*: A registered driver should call this function to remove a device that it added via `IPMAddDevice`.
  - *Input Arguments*:

    `DeviceID` – The device identifier assigned by the Policy Manager.

    `DriverHandle` – The driver handle assigned to the driver by the Policy Manager at the time of registration.
  - *Output Arguments*: None
  - *Return Values*:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

- IPMDevicePowerStateChanged
  - *Prototype*: IPMStatus IPMDevicePowerStateChanged(DriverHandle, DeviceID, NewState);
  - *Discussion*: A registered driver should call this function when it changes one its device's power state.
  - *Input Arguments*:

    `DriverHandle` – The driver handle assigned by the power manager software at the time of registration.

    `DeviceID` – The device identifier assigned by the power manager software at the time of device addition.

    `NewState` – Type `DevStatesEnum`, this represents the new state that the driver is transitioning its device to.
  - *Output Arguments*: None
  - *Return Values*:

    `IPMNoErr` – Indicates no error

    `IPMErr` – Indicates error

Example Power Manager Software for PXA27x processor

```
typedef enum {
    IPMErr = 0,
    IPMNoErr =1
} IPMStatus;

typedef enum {
```

```
    Run,
    M13,
    Standby,
    Sleep
} CPUState;

typedef struct _IPMState {
    CPUState CPUState;// Current Processor Mode
    UINT32 CPUVoltage;// Current Core Voltage * 100
    UINT32 CPUFrequency;// Current Core Frequency * 100
    UINT32 PXFrequency;// Current Bus Frequency * 100
    UINT32 SDClk0Frequency;// Current SDCLK0 Frequency * 100
    UINT32 SDClk1Frequency;// Current SDCLK1 Frequency * 100
    UINT32 SDClk2Frequency;// Current SDCLK2 Frequency * 100
} IPMState;

typedef enum {
    Core,
    MemClk,
    LcdClk
} ClksEnum;

const UINT32 FreqArray[][3] = {// Core, Mem, LCD
    62400, 20800, 10400,
    52000, 20800, 10400,
    41600, 20800, 10400,
    31200, 20800, 10400,
    20800, 20800, 10400,
    31200, 10400, 5200,
    15600, 10400, 5200,
    10400, 10400, 5200
};

typedef struct _FreqStruct {
    UINT32 Core;
    UINT32 Mem;
    UINT32 Lcd;
} FreqStruct;

typedef FreqStruct FreqArray [sizeof(BvdFreqArray)/sizeof(UINT32)/(LcdClk+1)];
```

```
typedef enum {
    Off,
    On,
    LowPower
} DevStatesEnum;

typedef struct _IPMDeviceInfo {
    char       DeviceId;// Device Identifier
    DevStatesEnumDeviceState;// Current State of Device
} IPMDeviceInfo;

typedef struct _IPMDriverInfo {
    USHORT        DriverHandle;// Id for a registered driver
    IPMDeviceInfo Devices[8];// Max of 8 devices
    char          NumDevices;// indicates #devices managed
    char          *CallBack;    // callback function
    DriverFreqSensitivity  FreqSensitivity;  // 32-bit bitfield
    DriverStateSensitvity  StateSensitvity;  // 32-bit bitfield
} IPMDriverInfo;

typedef UINT32  DriverFreqSensitivity; // bit field based on FreqArray
typedef UINT32  DriverStateSensitvity; // bit field based on CPUState
```

Summary

Marvell® Scalable Power Manager implements Scalable Power Manager in software. The PXA27x Processor Family implements Scalable Power Manager in hardware. Marvell® Scalable Power Manager software works with any operating system and the PXA27x processor to provide an infrastructure that uses power (voltage) and performance (frequency) scalability to dynamically fine-tune the power policy of a system.

Marvell Semiconductor, Inc.
5488 Marvell Lane
Santa Clara, CA 95054, USA
Tel: 1.408.222.2500
Fax: 1.408.752.9028
Email: commsales@marvell.com
www.marvell.com